

COMPUTER ENGINEERING

**BY
PROF. E. M. SAAD**

CONTENTS

Contents

1. Introduction	1
2. Design of CPU	7
2.1 Basic CPU Organization	7
2.2 Arithmetic Logic Unit (ALU)	11
2.2.1 Arithmetic Circuit	13
2.2.2 Logic Circuit	16
2.3 Floating-Point Arithmetic	18
2.3.1 Basic Operations	19
2.3.2 Basic ALU Organization	20
2.4 Control Circuit	22
Problems	23
3. Control Logic Design	26
3.1 Hardwired Control	27
3.1.1 State-table Method	27
3.1.2 Delay-element Method	28
3.1.3 Sequence-counter Method	31
3.1.4 Design example (for multiplier)	35
3.1.5 CPU Control Unit	43
3.2 Microprogrammed Control	47
3.3 Control of Processor Unit	49
3.3.1 Encoding of Microinstruction	51
Problems	56
4. Memory Systems	59
4.1 Memory Hierarchy	59
4.2 Standard Memory Systems	61
4.3 Cache Memory	64
4.4 Associative Memory	68
4.4.1 Hardware Organization	69
4.4.2 Read and Write Operations	70
4.5 Virtual Memory	71
4.5.1 Address Space and Memory Space	72
Problems	73
5. Register Transfer and Computer Operations	74
5.1 Register Transfer	74
5.1.1 Basic Symbols for Register Transfer	76
5.1.2 Multiplexer Selection	77
5.2 Bus Transfer	77

5.2.1 Three-state Bus Buffers	79
5.3 Memory Transfer	81
Problems	84
6. Input/Output and Communications	86
6.1 IO Techniques	86
6.1.1 Programmed IO	88
6.1.2 DMA and Interrupts	92
6.2 Input-Output Processor (IOP)	99
6.3 Handshaking	100
Problems	102
7. Parallel Processing	104
7.1 Multiple Processor Organization	105
7.2 Symmetric Multiprocessors	107
7.3 Cluster	112
7.4 Nonuniform Memory Access (NUMA)	115
Problems	118
Appendix	119
References	121

CHAPTER ONE

INTRODUCTION

Computing machines may be analog or digital. In this course, we are dealing with the hardware of digital computers.

The first successful attempts to devise automatic calculating machines appear to have been made in Europe in the seventeenth century.

In 1642, the French scientist Blaise Pascal built a mechanical adding machine. Around 1671 Gottfried Leibniz extended Pascal's machine to perform multiplication and division automatically. The mechanical calculator underwent further development in the second half of the nineteenth century (such as the use of electric motors to increase operating speed). Charles Babbage tried to build a computing machine capable of automatic multistep calculations (England in 1823). This machine, called a difference engine, was designed to compute table of functions such as logarithms and trigonometric functions. In the 1830's Babbage conceived of a much powerful mechanical computers which be called Analytical Engine.

The first successful general purpose digital computers were built in the 1830's, a full century after Babbage. In Germany, Konrad Zuse constructed several small mechanical computers. One of these, called the Z3, used electromechanical relays and binary instead of decimal number representation.

Vacuum tube technology underwent considerable development after the introduction of the triode in 1906. The first electronic digital computer employing vacuum tubes appears to have been a machine designed by John V. Atanasoff at Iowa state university in the late 1830's. This was a relatively small computer intended to solve up to 30 simultaneous linear equations (main computing mechanism was an add-subtract unit, that contained about 300 Vacuum tubes).

The first widely known electronic digital computer was the ENIAC (Electronic Numerical Integrator and Calculator) built at the univ. of Pennsylvania, (under the direction of John W. Mauchly and J. Prespert Eckert). The ENIAC was completed in 1946 and was used for a wide variety of computations until 1955. Physically it was very large, containing about 18,000 vacuum tubes and weighting about 30 tons.

A new electronic computer called the EDVAC (Electronic Discrete

Variable Automatic Computer) proposed by the ENIAC designers, including Mauchly, Eckert, and John von Neumann, embodied what is called the stored program concept, in which a single relatively large memory is used for storing both instructions and data during program execution (draft reported by Von Neumann in 1945). Even more influential was the design of a computer, now known as the IAS (Institute of Advanced Studies) computer which was built by Von Neumann and his colleagues in Princeton 1946-1952.

The IAS computer served as the prototype for most subsequent computers, and the term "Von Neumann Computer" has become synonymous with standard computer architecture.

Figure.1 shows the structure of a typical "first generation" computer of the IAS type.

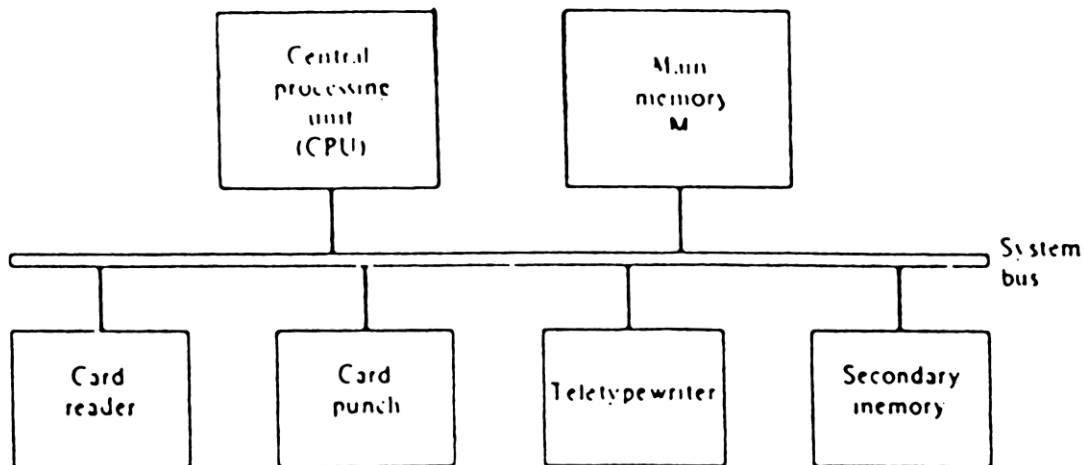


Fig.1 Organization of a typical 1st generation computer.

The CPU typically has two main parts: an instruction unit or I-unit responsible for fetching instructions and data from M, and execution unit or E-unit, which performs the action specified by the instructions.

In general, instructions of IAS-like machines fall into three main groups:

1. Data transfer instructions, that copy or move information without changing it from one part of the computer to another.
2. Data processing instructions, such as add, subtract, and compare, which transform data in a well-defined way.
3. Program-control instructions, such as conditional branch instructions, which determine the sequence in which instructions are executed.

Besides the CPU and M, a typical computer contains some IO or peripheral devices. The IO devices include secondary memories that serve as a backup memory for M. The various components of the computer communicate via a set of shared electrical cables referred to as a system bus. It is worth noting that the representative first generation computer organization depicted in Fig.1 is the same as that found in most current microcomputer systems.

Two other important characteristics of modern computers are: the use of binary or base-2 number representation, and the grouping of information into n-bit units called word.

In the early 1950's, there were also important developments in the methods used for producing computer programs, that is, computer software. The earliest computers were programmed exclusively in machine language, which is binary code that can be executed directly by the CPU. A significant reduction in programming difficulty resulted from the introduction of assembly languages, which are programming languages that allow instructions to be written in symbolic form using easily remembered names for the operations and operands of each instruction. A special program called an assembler is required to translate the assembly language (source) program into a machine-language (object) program before it is executed.

In the 1950's transistors slowly began to replace vacuum tubes as the main electronic components in CPU design. One of the earliest second generation or transistor computer was the TX-O (experimental machine built at the Massachusetts Institute of Technology in 1953).

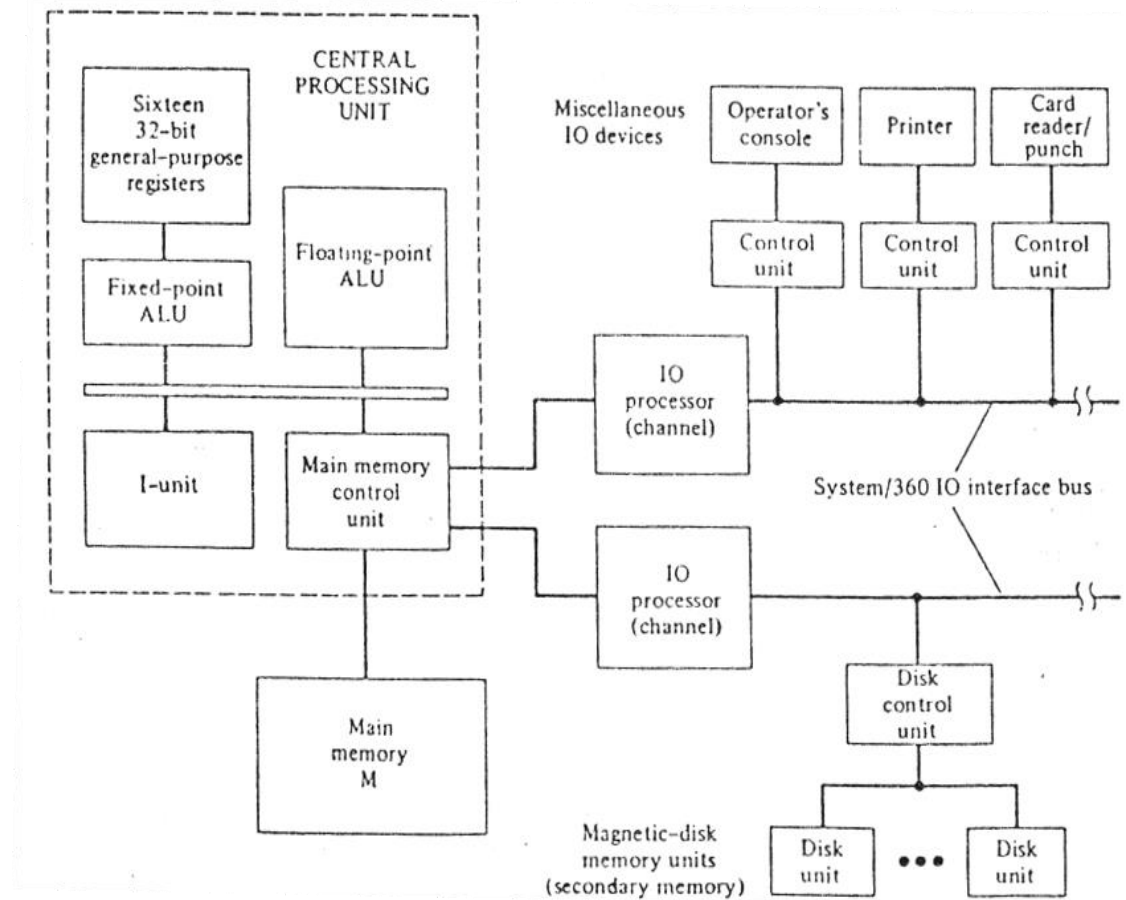
As computer usage increased, the difficulty of writing programs become a major cause for concern.

In the mid-1950's another class of programming languages, called the high-level languages was introduced.

In 1964, IBM introduced a very influential computer series called the system/360. Although often considered to be the earliest third generation or IC-based computer family, the system/360 does not use ICs, instead it employs thick-film circuits on which the discrete transistors are mounted. In many system/360 models the I-unit is microprogrammed, which means that its actions are specified by a set of low-level programs called microprograms stored in a special memory, the control memory, within the I-unit itself. Microprogramming greatly enhances the flexibility of a CPU, making it relatively easy for the manufacture to make addition and changes the CPU's instruction set. Fig.2 shows the organization of a typical third generation computer.

Most computers manufactured around 1960 were large and costly mainframe machines. Much of this cost was for computer hardware, with the CPU and M among the more costly items. Considerable effort was devoted to developing ways of using the available computer resources as efficiently as possible.

An important step in this direction was the development of the operating system, a supervisory program usually applied by the computer manufacturer that allows many users to use the various resources of a computer concurrently.



In 1965, Digital Equipment Corporation (DEC) introduced the PDP-8 (Programmed Data Processor Eight), one of the first of a class of small computers now referred to as minicomputers. Minicomputers are characterized by a short CPU word size (12-bits in PDP-8). An IC version for PDP-8 was introduced in 1967. By 1970 small scale and medium-scale ICs were found in almost all new computers. At that time, MOS LSI chips was also being mass produced, mainly for use in computer main memories

and packet calculators. The stage was set for the next major development, the one chip CPU, or microprocessors.

The functional capabilities of microprocessors and microcomputers have continued to increase rapidly with improvements in LSI/VLSI technology. Microprocessors with CPU word sizes of 16 and 32 bits have appeared, for example, the Motorola 68000.

Microprocessors and microcomputers have been responsible for a huge increase in the production of general-purpose computers, most of which have been used in new applications such as word processing and personal computing. The computer design process is the topic of this course. It is viewed as having three major levels of complexity: the gate, register, and processor levels. The two main aspects of gate-level design, combinational, and sequential logic design, as well as the design at the register level, are reviewed in the 4th year course. The design at processor level is discussed.

CHAPTER TWO

DESIGN OF CPU

The component types recognized at the processor level fall into four main groups: (instruction set) processors, memories, input/output devices, and interconnection networks. For obvious reasons, information is transferred between the components in words or small groups of words.

2.1 Basic CPU Organization

The central processing unit (CPU) is defined to be a general-purpose instruction set processor. It is the central component of a digital computer. The purpose of the CPU, is to interpret instruction codes received from memory and perform arithmetic, logic, and control operations with data stored in internal registers, memory words, or I/O interface units. Externally, the CPU provides a bus system for transferring instructions, data, and control information to and from the modules connected to it.

A typical CPU is usually divided in two parts: the processor unit and the control unit. The processor unit consists of an arithmetic logic unit (ALU), a number of registers, and internal buses that provide the data paths for the transfer of instruction between the register and the ALU. The control unit consists of a program counter, an instruction register, and timing and control logic. The control logic may be either hardwired or microprogrammed.

If it is hardwired, registers, decoders, and a random set of gates are connected to provide the logic that determines the actions required to execute the various instructions. A microprogrammed control unit uses a control memory to store microinstructions and a sequencer to determine the order by which the microinstructions are read from control memory.

Generally, only one CPU is present in a computer installation. A computer with one CPU is called a uniprocessor to distinguish it from a multiprocessor, which has two or more CPUs.

Despite the great improvements in circuit technology over the years, almost all CPU designs have been based on the following two premises:

1. The CPU should be as fast (measured by its cycle time t_{cpu}) as the available technology permits. Since cost in the CPU must be kept relatively small.

2. A main memory of relatively large capacity is needed to store the programs and data required by the CPU.

The design proposed by Von Neumann and his colleagues for the IAS computer is the basis for almost all CPUs designed since then. It comprises a minimal set of registers and the necessary circuits to execute a small single-address instruction set. One of the CPU registers, called the accumulator, plays a central role, being used to store an input or an output operand (result) in the execution of most instructions.

Figure 3 shows the essential structure of an accumulator-oriented CPU. This architecture is typical of 1st-generation machines such as IAS, and some modern computers in the mini and micro classes. The accumulator AC is the main operand register of the arithmetic-logic unit. The data register DR acts as a buffer with the accumulator so that operations of the form $AC-f(AC,DR)$ can be performed. The other major registers are the program counter IR, which holds the opcode of the current instruction, and the memory address register AR.

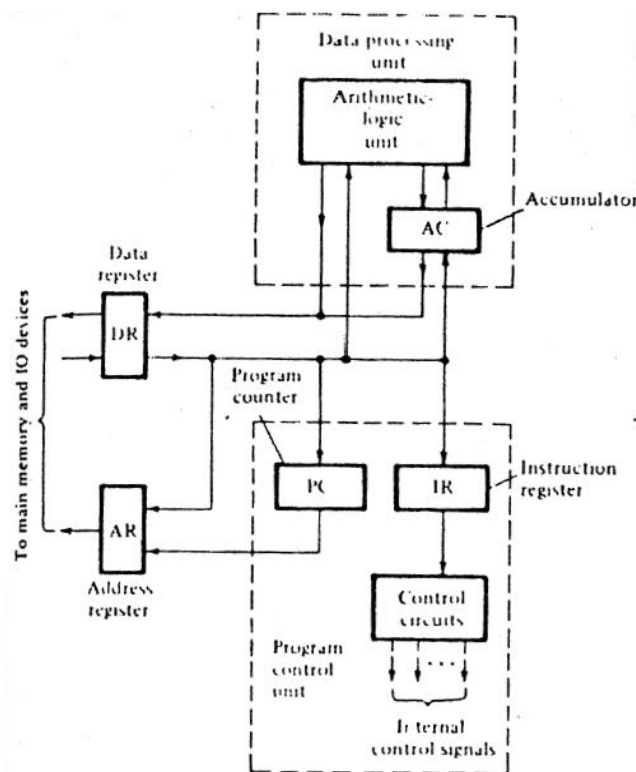


Fig.3 A simple accumulator-based CPU

In addition to executing programs, the CPU supervises the other system components, usually via special control lines.

For example, the CPU directly or indirectly controls IO operations such as data transfers between IO devices and main memory. These operations require the CPU attention relatively infrequently; it is therefore more efficient to allow the CPU to ignore IO devices and the like until they actively request service from the CPU. Such a request is called interrupt. In the event of an interrupt, the CPU suspends execution of the program that it is executing and transfers to appropriate interrupt handling program. Interrupts, particularly IO interrupts, frequently require a rapid response by the CPU. A test for the presence of interrupt signals is thus normally carried out. At the end of each instruction cycle. The major functions of the CPU are summarized in the flowchart in Fig.4.

There are several ways in which the basic processor organization of Fig.3 can be made more powerful.

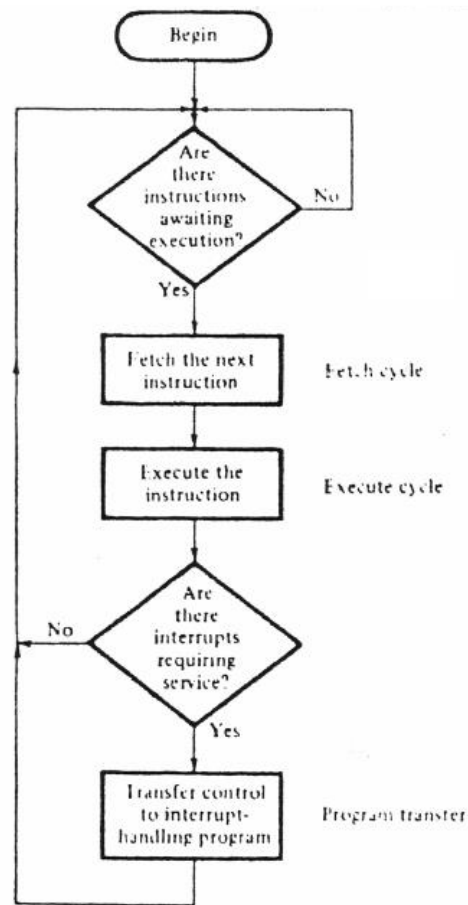


Fig.4 Overview of CPU hardware.

1. Additional registers can be provided for storing operands and addresses, (as replacing the single accumulator by a set of registers).
2. Extending the arithmetic capabilities.
3. Special registers can be included to facilitate the transfer of control between instructions within a program.
4. The transfer of control between different subprograms (subroutines or procedures) due to interrupts and subroutine calls and returns is also facilitated by special registers.
5. Facilities can be provided for simultaneous processing of two or more distinct instructions.

Fig.5 shows the register-level design of a typical CPU based on the

forgoing considerations.

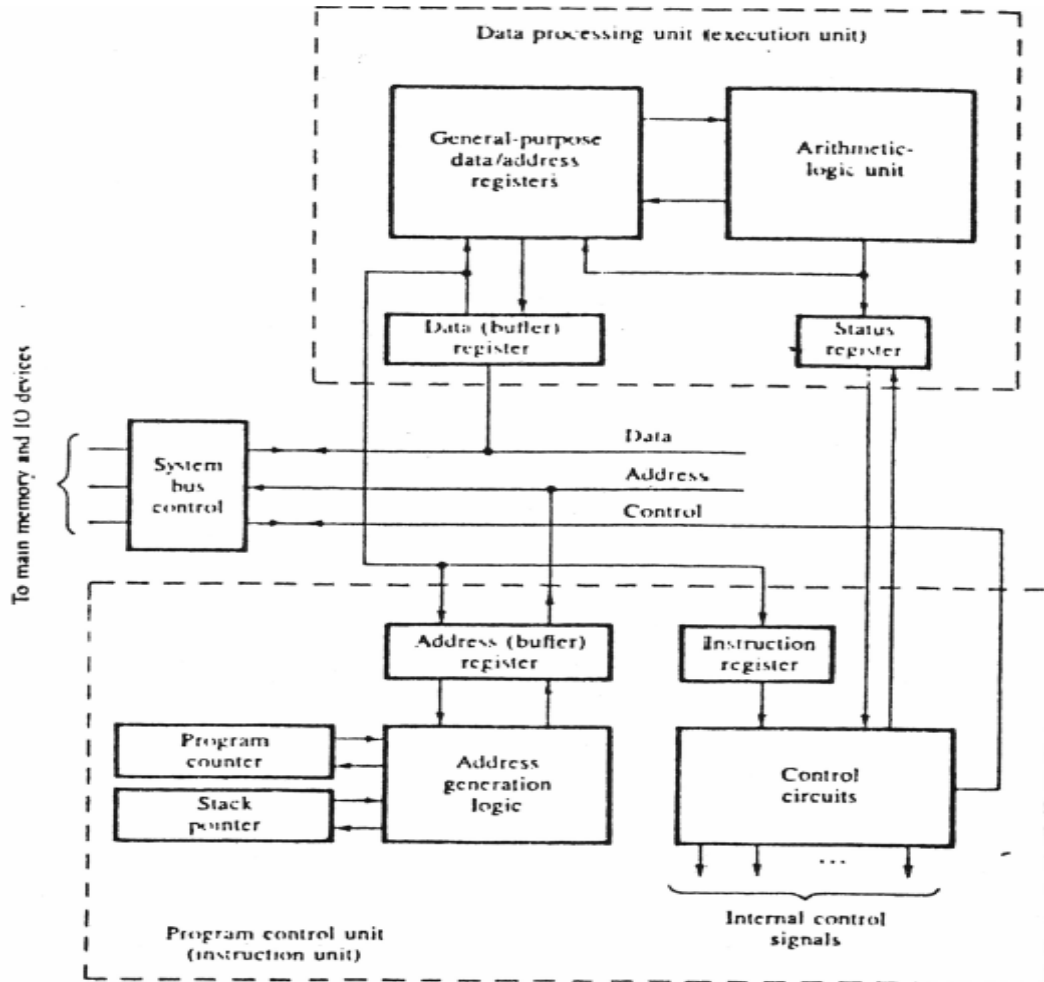


Fig.5 Typical CPU with general register organization.

2.2 Arithmetic Logic Unit (ALU)

An arithmetic logic unit (ALU) is a combinational circuit that performs a set of basic arithmetic and logic microoperations. The ALU has a number of selection lines used to select a particular operation in the unit. The selection lines are decoded within the ALU so that K selection variables

can specify up to 2 distinct operations.

Fig.6 shows the block diagram of a typical 4-bit ALU.

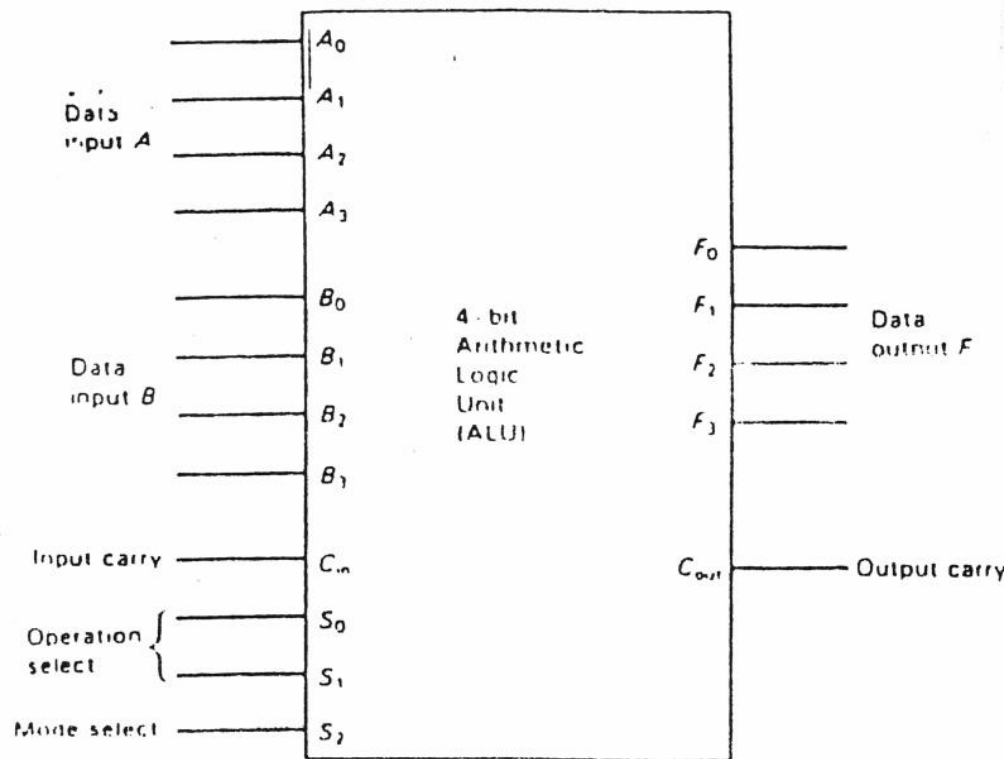


Fig.6 Block diagram of a 4-bit ALU.

The four data inputs from A are combined with the four data inputs from B to generate an operation at the F outputs. The mode select input S distinguishes between arithmetic and logic operations.

The two function select inputs S and S specify the particular arithmetic or logic operation to be generated. The input and output carries have meaning only during an arithmetic operation.

The input carry C is quite often used as a fourth selection variable for arithmetic operations. In this way, it is possible to double the number of arithmetic operations from four to eight.

The design of a typical ALU will be carried out in three stages. First, the design of the arithmetic section will be undertaken. Second, the design of the logic section will be presented. The two sections will then be combined to form the arithmetic logic unit.

2.2.1 Arithmetic Circuit

The basic component of an arithmetic circuit is the parallel adder. A parallel adder is constructed with a number of full-adder circuits connected in cascade (4th year course). By controlling the data inputs to the parallel adder, it is possible to obtain different types of arithmetic operations.

The block diagram of Fig.7 demonstrates a possible configuration where one set of inputs to the parallel adder is controlled by two selection lines S_1 and S_0 . There are n -bits in the arithmetic circuit with two inputs, A and B , and one output F . The n -inputs from B go through a combinational circuit to the y -inputs of the parallel adder.

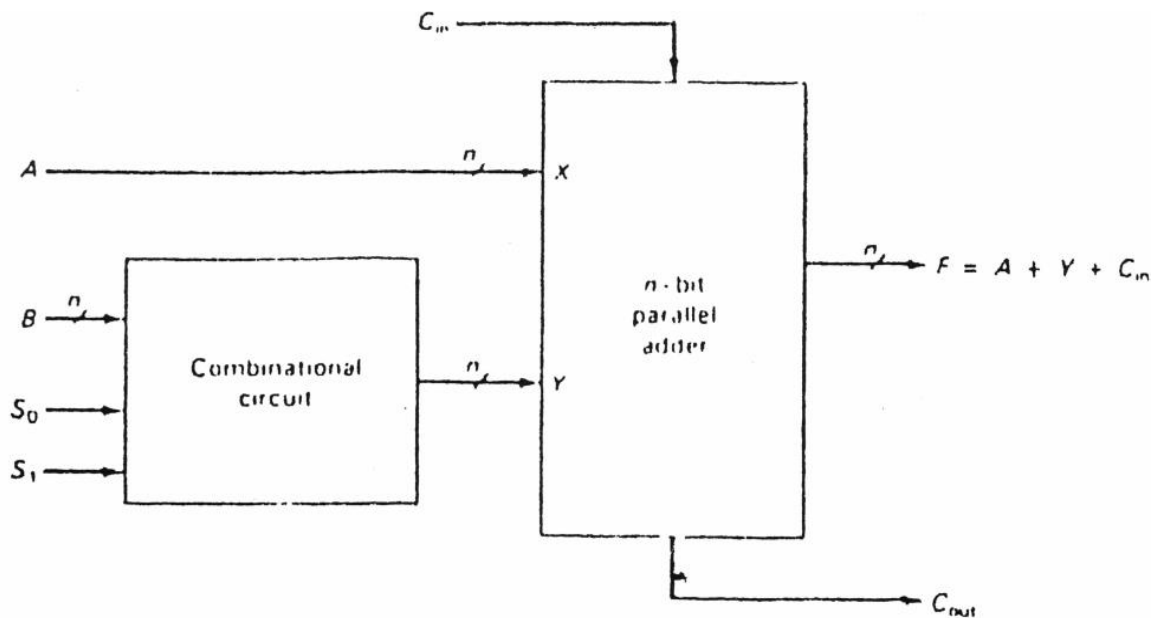


Fig.7 Block diagram of an arithmetic circuit.

The input carry C goes to the carry input of the full-adder circuit in the least significant position. The output of the parallel adder is calculated from

the following arithmetic sum:

$$F = A + Y + C, \quad \text{where } C \text{ can be equal 0 or 1.}$$

Note that the symbol (+) in the above equation denotes an arithmetic plus.

By controlling the value of the Y with S_1 and S_0 , it is possible to obtain a variety of arithmetic operations. This is shown in the following table.

Arithmetic Circuit Function Table

Select $S_1 S_0$	Input Y	F = A + Y + C	
		$C_{in} = 0$	$C_{in} = 1$
0 0	all 0's	F = A(Transfer)	F = A+1(increment)
0 1	$\underline{B}$	F = A+B(add)	F = A+B+1
1 0	$\overline{B}$	F = A+B	F = A+B+1(subtract)
1 1	all 1's	F = A-1(decr.)	F = A(transfer)

If the inputs from B are ignored and we insert all 0's into the Y inputs, the output becomes $F = A + 0 + C$. This gives:

$$\begin{aligned} F &= A & \text{when } C_{in} &= 0. \\ F &= A + 1 & \text{when } C_{in} &= 1. \end{aligned}$$

In this first case, we have a direct transfer from A to output F. In the second case, the value of A is incremented by 1. For a straight forward addition, it is necessary to transfer the B inputs into the Y inputs of the parallel adder. This gives $F = A + B$ when $C_{in} = 0$.

Arithmetic subtraction is achieved when $Y = B$ to obtain $F = A + \overline{B} + 1$ when $C_{in} = 1$.

This gives A plus the 2's complement of B which is equivalent to subtraction. Inserting all 1's into the inputs of Y gives $F = A - 1$ when $C_{in} = 0$. When $C_{in} = 1$, transfer of A occurs.

The combinational circuit of Fig.7, can be implemented with n

multiplexers. The data input to each multiplexer in stage $i = 0, 1, 2, \dots, n-1$ are : 0 , B_i , B_i , and 1, corresponding to selections $S_1 S_0 : 00, 01, 10$, and 11 respectively. Thus, the arithmetic circuit can be constructed with n multiplexers and n Full-adders.

The number of gates in the combinational circuit can be reduced if, instead of using multiplexers, we go through the logic design of one stage of the combinational circuit. This can be done as in the following table.

		S1 S0			
		00	01	11	10
Bi	0			1	1
	1		1	1	

K-map

Inputs			Output	
S1	S0	Bi	Yi	
0	0	0	0	$Y_i = 0$
0	0	1	0	
0	1	0	0	$Y_i = B_i$
0	1	1	1	
1	0	0	1	$Y_i = \overline{B_i}$
1	0	1	0	
1	1	0	1	$Y_i = 1$
1	1	1	1	

Truth Table

$$Y_i = B_i S_0 + \overline{B_i} S_1$$

The logic diagram of a 4-bit arithmetic circuit is shown in Fig.8. The four full adder (FA) circuits constitute the parallel adder. C_{in} adds 1 to the sum when it is equal to 1.

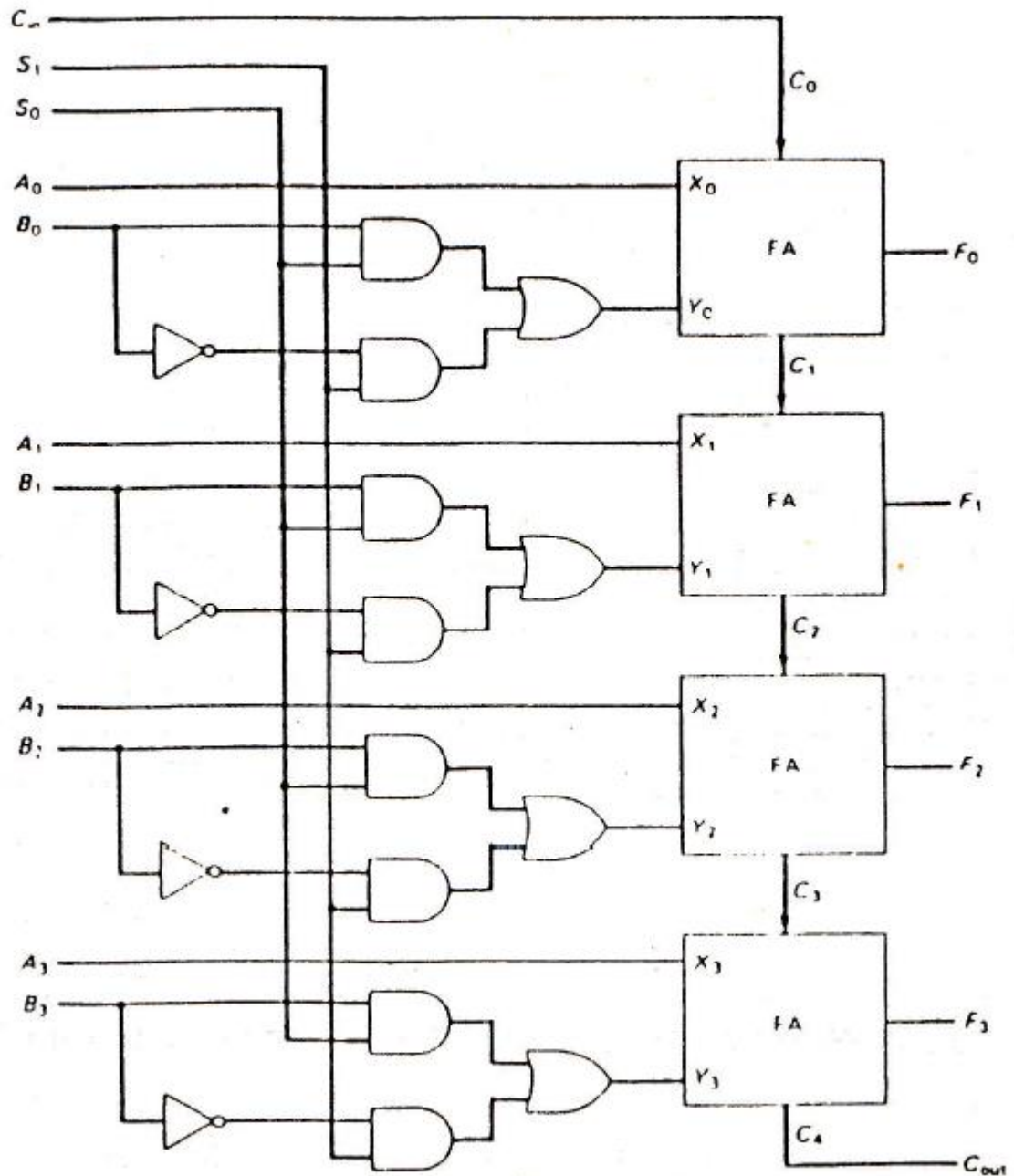


Fig.8 Logic diagram of an 4-bit Arithmetic circuit.

2.2.2 Logic Circuit

The logic microoperations manipulate the bits of the operands by

treating each bit in a register as a binary variable. There are four basic logic operations from which all others can be derived. They are the AND, OR, XOR and the complement. Fig.9 shows one stage of the logic circuit. The diagram shows one typical stage with subscript i. For logic circuit with n-bits, the diagram must be repeated n times.

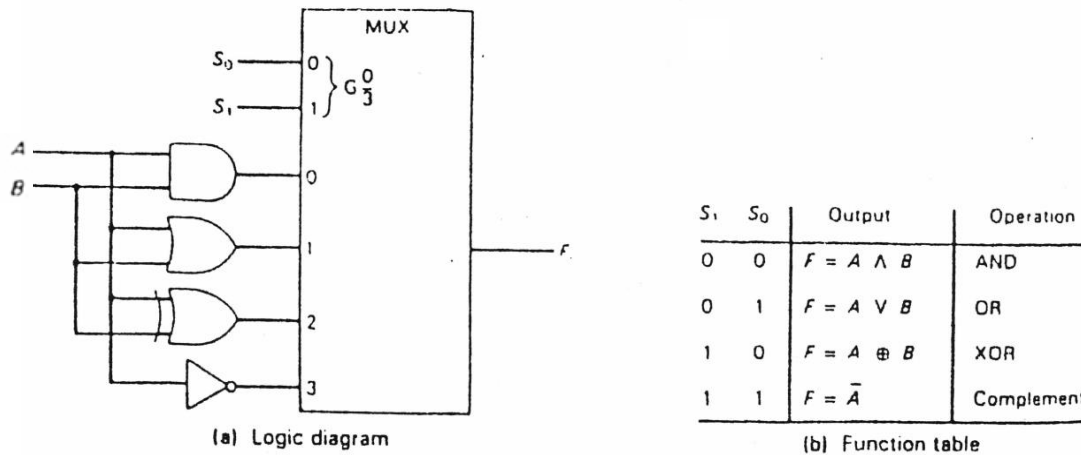


Fig.9 One stage of the logic circuit.

The logic circuit can be combined with the arithmetic circuit to produce an arithmetic logic unit (ALU). Selection variables S_1 and S_0 can be common to both circuits provided we use a third selection variable to differentiate between the two. The configuration of the ALU is illustrated in Fig.10.

The circuit of Fig.10 must be repeated n times for an n-bit ALU. The output carry C_{i+1} of a given stage must be connected to the input carry C of the next stage in sequence.

The input carry to the first stage is the input carry C_{in} which provides a selection variable for the arithmetic operations.

The ALU specified in Fig.10, provides eight arithmetic and four logic operations. The input carry C_{in} is used for selecting an arithmetic operation only.

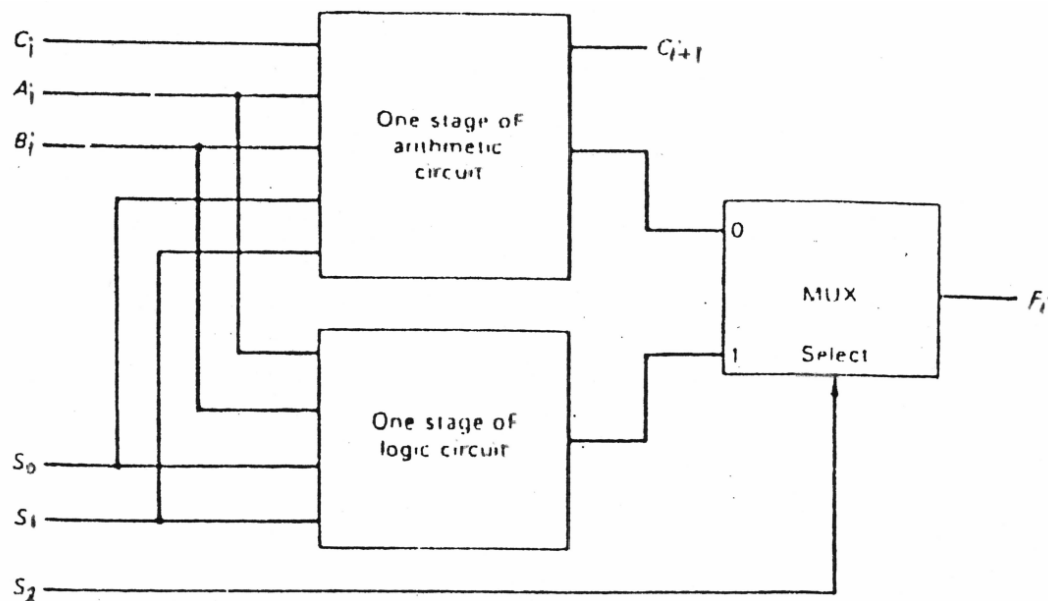


Fig.10 One stage of an ALU.

The following table lists the 12 operations of that ALU.

Operation select				Operation	Function
S_2	S_1	S_0	C_{in}		
0	0	0	0	$F = A$	Transfer A
0	0	0	1	$F = A + 1$	Increment A
0	0	1	0	$F = A + B$	Addition
0	0	1	1	$F = A + B + 1$	Add with carry
0	1	0	0	$F = A + \bar{B}$	A + 1'compl of B
0	1	0	1	$F = A + \bar{B} + 1$	Subtraction
0	1	1	0	$F = A - 1$	Decrement A
0	1	1	1	$F = A$	Transfer A
1	0	0	0	$F = A \text{ (AND) } B$	AND
1	0	1	0	$F = A \text{ (OR) } B$	OR
1	1	0	0	$F = A \oplus B$	XOR
1	1	1	0	$F = \bar{A}$	Complement A

2.3 Floating-point Arithmetic

Let (X_M, X_E) be the floating-point representation of a number X.

Hence $X = X_M \cdot B^{X_E}$. It is also assumed that floating-point numbers are stored in normalized form only, hence the results of all floating-point arithmetic operation should be normalized.

2.3.1 Basic Operations

The formulas to perform floating-point addition, subtraction, multiplication, and division are given in the following table.

Addition	$X + Y = (X_M \cdot 2^{X_E - Y_E} + Y_M) \cdot 2^{Y_E}$	where $X_E < Y_E$
Subtraction	$X - Y = (X_M \cdot 2^{X_E - Y_E} - Y_M) \cdot 2^{Y_E}$	
Multiplication	$X * Y = (X_M * Y_M) \cdot 2^{X_E + Y_E}$	
Division	$X / Y = (X_M / Y_M) \cdot 2^{X_E - Y_E}$	

Multiplication and division are not significantly more difficult to implement than the corresponding fixed-point operations. Floating-point multiplication requires a fixed-point multiplication of the mantissas and a fixed-point addition of the exponents. Floating-point division requires a fixed-point division involving the mantissas and a fixed-point subtraction involving the exponent.

Floating-point addition and subtraction are complicated by the fact that the exponents of the two input operands must be equal before the corresponding mantissas can be added or subtracted.

As suggested in the table, this can be done by right-shifting the mantissa X_M associated with the smaller exponent X_E a total of $Y_E - X_E$ digit positions to form a new mantissa $(X_M \cdot 2^{X_E - Y_E})$ which can then be combined with Y_M . Thus, addition and subtraction require the following three steps:

1. Compute $Y_E - X_E$ (a fixed point subtraction).
2. Shift X_M , $Y_E - X_E$, places to form $(X_M \cdot 2^{X_E - Y_E})$.
3. Compute $(X_M \cdot 2^{X_E - Y_E}) \pm Y_M$ (fixed-point addition or subtr.).

In general, the way to expand the range of numbers is to express them in floating-point notation. Decimal floating-point numbers are interpreted as represented a number in the form $F \cdot 10^E$, where F is the fraction (mantissa) and E is the exponent. Only the fraction and the exponent are physically represented in computer registers.

The base 10 and the decimal point of the fraction are assumed and are not shown explicitly.

A floating-point binary number is represented in a similar manner except that it uses a base 2 for the exponent. For example, the binary number + 1001.11 is represented with an 8-bit fraction and 6-bit exponent as:

Fraction (Mantissa)	Exponent
01001110	000100

The fraction has a 0 in the leftmost position to denote a plus. The binary of the fraction follows the sign bit but is not shown in the register.

The exponent has the equivalent binary number +4. The floating-point number is equivalent to:

$$F \cdot 2^E = + (.1001110)_2 * 2^{+4}$$

A floating-point number is said to be normalized if the most significant digit of the fraction is nonzero. (for ex. 0.350 is normalized, but 0.0035 is not normalized).

2.3.2 Basic ALU Organization

Fig.11 shows one of the most widely used ALU designs, which aims at minimizing hardware costs. Floating-point arithmetic can be implemented by two loosely connected fixed-point arithmetic circuits, an exponent unit and a mantissa unit as suggested in Fig.12.

The mantissa unit is required to perform all four basic operations on the mantissa, hence a general-purpose fixed-point arithmetic unit such that of Fig.11 can be used.

A simpler circuit capable of only adding, subtracting, and comparing exponents suffices for the exponent unit.

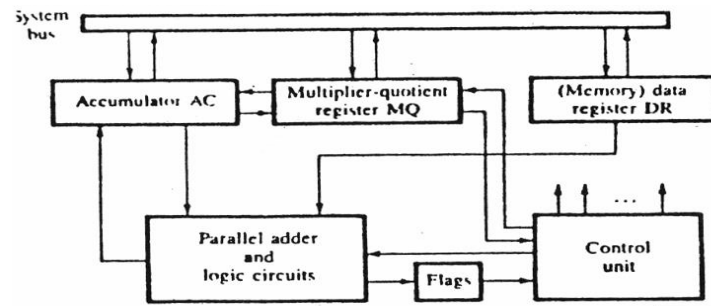


Fig.11 Structure of a basic fixed-point ALU.

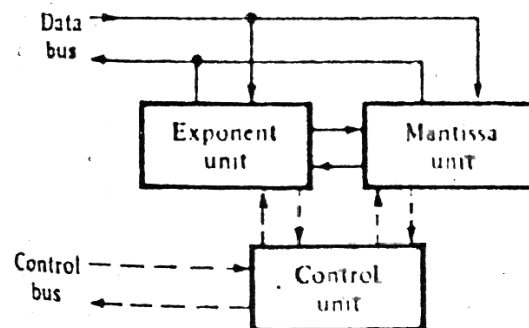


Fig.12 A floating-point arithmetic unit viewed as two fixed-point arithmetic units.

2.4 Control Circuit

The hardware of the control unit consists of the control memory that stores the microprogram, the address sequencing logic that chooses the next microinstruction address, and decoder and multiplexers that select the various components of the microoperation. The part of the control unit that determines the address of the next microinstruction is called the microprogram sequencer. In addition to the sequencer, the control must incorporate the hardware for decoding certain fields of the microinstruction. In the next chapter, the control logic design is presented.

Problems

[1]

- a. For the logic circuit shown in the figure below:
Derive an expression for the output in a POS form. Implement the circuit using NAND gates only.
- b. Perform the following operations:
 $25 * 9$ (using two methods), $7/5$, and $(125+90)_{10}$ in BCD.

[2]

- a. Reduce the following Boolean expression (using K-map):
$$F = XY + XZ + XYZ + YZ$$
- b. Design a combination circuit with 4-inputs and one output. The output should be logic 1, when the binary input is 4 and its multiples, and logic 0, otherwise.

[3]

- a. Describe with the aid of an application example a combinational and a sequential logic circuit.
- b. A sequential circuit has two JK-FF, one input X and one output Y. The Flip-Flop input equations and output equation are:

$$J = K = \overline{B}, J = K = \overline{A + X} \text{ \& } Y = B \oplus (A \oplus X)$$

Draw the logic diagram of the circuit. Derive the state table and the state diagram for such a circuit.

[4] Design a radix 10 synchronous counter, starting with the count 4. Show the output waveforms.

[5] Explain how an accumulator differs from an adder.

[6] State the four basic functions performed by the CPU of a digital computer.

[7] Explain the difference between data and information.

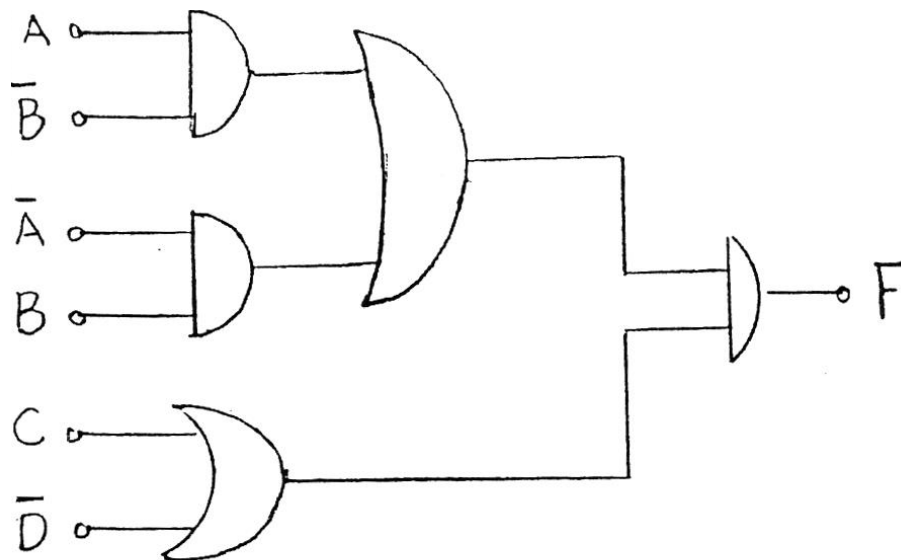
[8] Do the following arithmetic operations using scientific notation:

- a. $73.46 \times 10^{-4} - 0.00132 \times 10^{-2}$

- b. $6.93 \times 10^6 + 43.81 \times 10^5$
- c. $(83.76 \times 10^4) \times (154.8 \times 10^2)$
- d. $(724.89 \times 10^{12}) / (367.12 \times 10^6)$

[9] Design the arithmetic circuit with one selection variable's and two n-bit data inputs A & B. The circuit generates the following four arithmetic operations in conjunction with the input carry C. Draw the circuit diagram for the 1st two stages:

S	Cin = 0	Cin = 1
0	$F=A+B$ (add)	$F=A+1$ (increment)
1	$F=A-1$ (decrement)	$F=A+B+1$ (subtract)



P-1

Problems

- [1] Show how the following two floating-point numbers are to be added to get a normalized result:

$$(-1.3567 \times 10^{+3}) + (+67430 \times 10^{-1})$$

- [2] A 36-bit floating-point number consists of 26 bits plus sign for the fraction and 8-bits plus sign for the exponent. What are the largest and smallest +ve quantities for normalized numbers?
- [3] Show the logic diagram of a 4-bit arithmetic circuit that perform the operation in the table page (14) after Fig.7 in the lectures. Use four multiplexers and four full adders.
- [4] Design a four - bit arithmetic circuit with two selection variables S₁ and S₀ that generates the following arithmetic operations. Draw the logic diagram of one typical stage.

S ₁ S ₀	C _{in} =0	C _{in} =1
0 0	F=A+B	F=A+B+1
0 1	F=A	F=A+1
1 0	F=B	F=B+1
1 1	F=A+B	F=A+B+1

- [5] Design an arithmetic circuit with one selection variable S and two data inputs A and B. When S = 0, the circuit performs the addition operation $F = A + B$. When S=1, the circuit performs the increment operation $F = A + 1$.

CHAPTER THREE

CONTROL LOGIC DESIGN

The logic design of a digital system can be divided into two distinct parts. One part is concerned with the design of the digital circuits that performs the data processing operations. The other part is concerned with the design of the control circuit that supervises the operations and determines the sequence in which they are executed. The relationship between the control logic and the data processor subsystems in a digital system is shown in Fig.13.

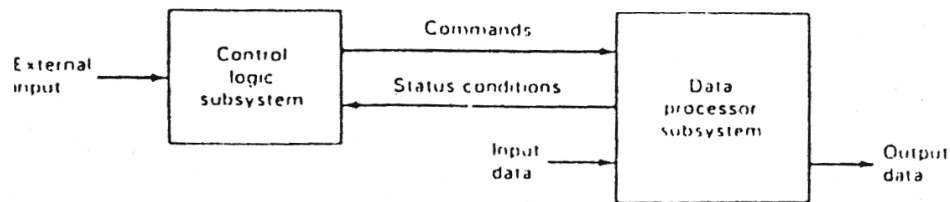


Fig.13 Control and data processor interaction.

The control logic initiates properly sequenced command signals to the data processor. The control logic uses status conditions from the data processor serve as decision variables for determining the sequence of the controlled operations. The control logic that generates the signals for sequencing the operations in the data processor is essentially a sequential circuit with internal states that dictate the control commands for the system.

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates and flip-flops similar to a sequential circuit. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information such as control functions or control words are stored as 1's and 0's in a special memory.

3.1 Hardwired control

The design of a hardwired control unit involves various complex tradeoffs between the amount of hardware used, its speed of operation, and the cost of the design process itself. Because of the large number of control signals used in a typical CPU and their dependence on the particular instruction set being implemented, the design methods employed in practice are often adhoc and heuristic in nature, and therefore cannot easily be formalized. To illustrate the main issues involved, we consider three simplified and systematic approaches to the design of hardwired controllers. These methods are representative of those used in practice, but by themselves are suitable only for small control units such as might be encountered in nonprogrammable controllers or Reduced Instruction Set Computer (RISC) processors.

Method 1: The standard algorithmic approach to sequential circuit design, which is called the state - table method, since it begins with the construction of a state table for the control unit.

Method 2: A heuristic method based on the use of clocked delay elements for control-signal timing.

Method 3: A related method that uses counters, which we call sequence counters, for timing purposes.

Method 1, the most formal of these design approaches, may incorporate systematic techniques for minimizing the number of gates and flip-flops.

Method 2 and 3, which are less formal, attempt to derive a logic circuit directly from the original (flowchart) description of the control unit behavior. The resulting designs may not contain the minimum number of gates and flip-flops, but they are often obtained with much less effort.

3.1.1 State-table method:

A state-table description is a suitable starting point for the implementation of small control units. A well-defined design methodology exists using the state-table approach. There are several practical disadvantages to using state tables:

1. The number of state and input combinations may be so great that the state-table size and the amount of computation needed become excessive.
2. State tables tend to conceal useful information about a circuit behavior, e.g. the existence of repeated pattern or loops.

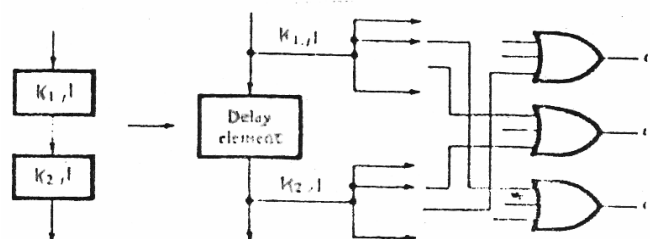
Control circuits designed from state tables also tend to have a random structure which makes design debugging and subsequent maintenance of the circuit difficult.

3.1.2 Delay-element Method

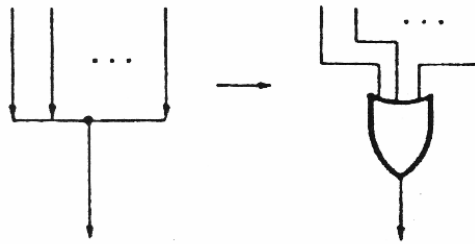
A control unit using delay elements can be constructed directly from a flowchart that specifies the control-signal sequence required. The circuit thus formed has essentially the same structure as the flowchart, a consequence of the fact that the circuit simply mirrors the flow of control through the flowchart. A few simple rules illustrated in the following figures, indicate the way in which the control circuit is derived from the flowchart.

1. Each sequence of two successive microoperations requires a delay element.

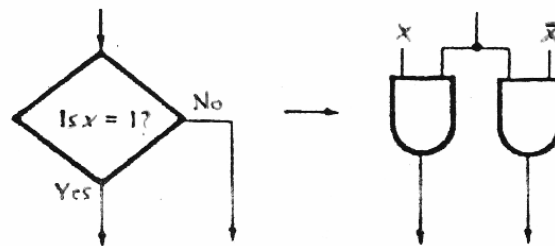
The signals that activate the control lines are taken directly from the input and output lines of the delay. Signals that are intended to activate the same control line C_1 are fed to an OR gate whose output is C_1 . This line may then be connected to the control point it activates.



2. k-lines in the flowchart that merge to a common line are transformed into a k-input OR gate.



3. A decision box (which indicates a branch in the control flow based on a condition test) can be implemented by two AND gates as shown.



This AND circuit forms a simple 1-bit demultiplexer controlled by the test variable x . Note that x may be replaced by a Boolean function $f(x)$, so that condition tests of arbitrary complexity can be used to determine the flow of control.

A portion of a typical flowchart indicating the control signals $[C]$ that must be activated at each step is shown. The control circuit obtained using these transformation rules is shown in Fig.14.

Note that the AND gates derived from the two decision boxes have been merged in an obvious manner. If all delays are one clock period in duration, then a clocked D-type master slave flip-flop can be used to construct the delay element, as shown in Fig.14 .

The control pulses are assumed to be of about the same duration as the clock pulse .A more complex circuit may be required if an input control pulse can become appreciably out of phase with the clock due to propagation delays between delay elements . This design method has the disadvantage that the number of delay elements needed is approximately equal to the number of states n_s . Furthermore, each delay element is a sequential circuit of equal or greater complexity than a flip-flop. With classical state-table design method, one can design a synchronous sequential circuit of n_s states with no more than $\log_2 n_s$ flip-flops.

Thus, the delay element approach tends to produce expensive circuits, with difficult synchronization.

3.1.3 Sequence-counter Method

Consider the following circuit of Fig.15, which consists basically of a modulo-k counter, whose output is connected to a $1/k$ clocked decoder. If the count enable input is connected to a clock source, the counter cycles continually through its k-states. The decoder generates k-pulse signals $[\phi_i]$, on its output lines. Two additional input lines and a flip-flop are provided for turning the counter on and off. A pulse on the begin line causes the counter to begin cycling, by logically connecting the count enable line to the clock source. A pulse on the end line disconnects the clock and resets the counter.

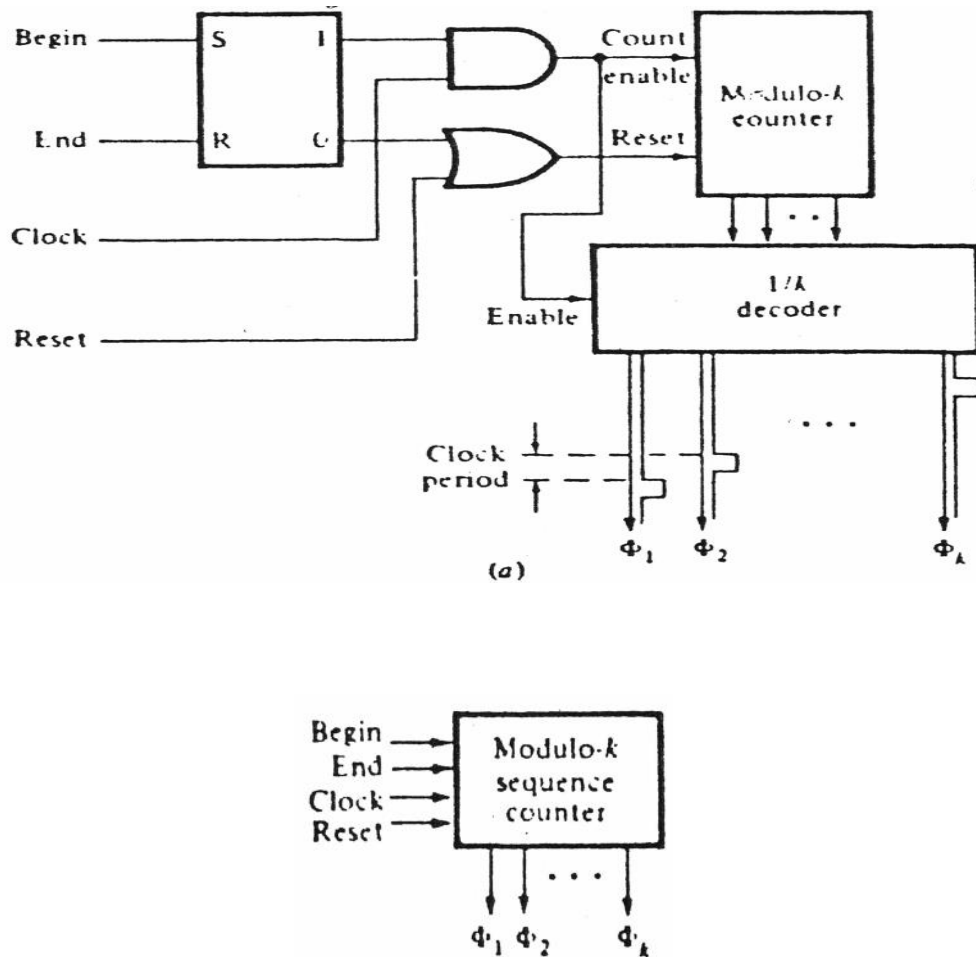
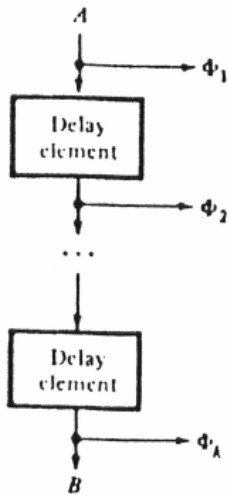


Fig.15 A modulo-K sequence counter logic diagram.

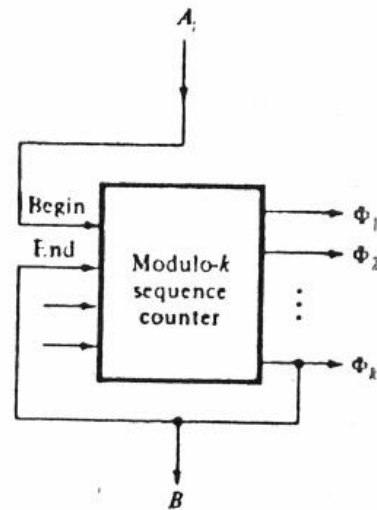
The usefulness of control counters of this type stems from the fact that many digital circuits are designed to perform a relatively small number of actions repeatedly. This type of behavior can be described (usually at a fairly high level) by a flowchart consisting of a single closed loop containing k -steps. For example Fig.16 shows a one-loop flowchart containing six steps that describes the behavior of a typical CPU. One may then build a control unit for this CPU around a single (modulo-6) sequence counter.

A strong relationship exists between the sequence counter and delay-element methods. A modulo- k sequence counter can easily be made to

behave like a cascade of $k-1$ delay elements. The counter shuts itself off after one complete cycle. Thus the control unit design method with delay elements can, in principle, be directly modified to apply to sequence counters if every cascade of $(k-1)$ delay elements is replaced by the following circuit. However, the resulting design would generally be very inefficient compared with a sequence counter design.



Delay-element cascade.



Equivalent sequence counter circuit.

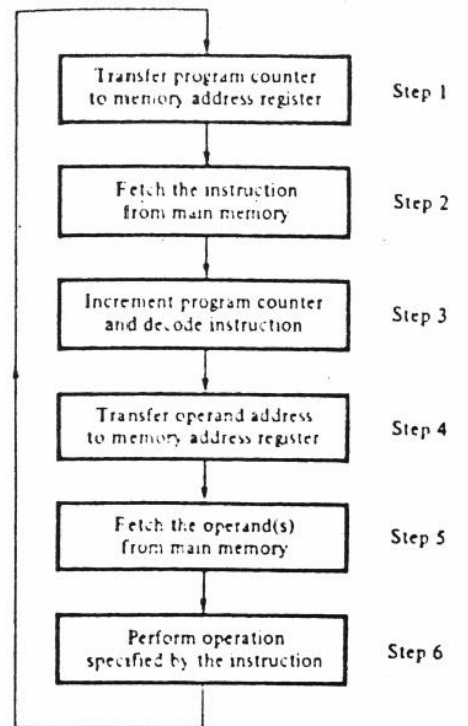


Fig.16 CPU behavior represented as a single closed loop.

Conversely, a cascade of $k-1$ delay elements can be made to behave like a sequence counter by connecting its output to the input via an additional delay-element and an OR gate as shown in Fig.17. The resulting circuit, which behaves like a free-running modulo- k sequence counter, is called a module- k ring counter. It is a useful component for control design.

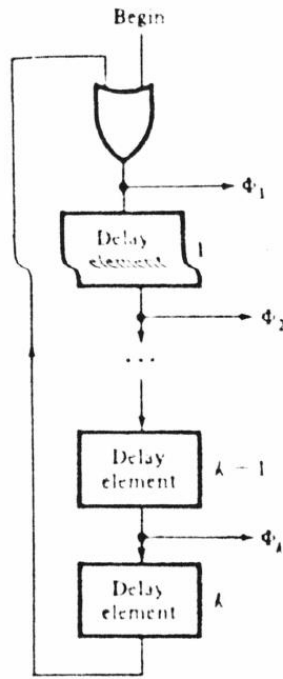


Fig.17 A delay-element circuit (ring counter) that behaves like a sequence counter.

3.1.4 Design Example: Hardwired control for multiplier

A flowchart showing the sequence of operations in the binary multiplier is shown in Fig.18. Initially, the multiplicand is in B and the multiplier in Q. The multiplication process starts when S becomes 1. Register A and flip-flop C are reset to 0 and the sequence counter P is loaded with a binary number n, which is equal to the number of bits in the multiplier.

Next we enter a loop that keeps the partial products. The multiplier in Q is checked, if it is equal to 1, the multiplicand in B is added to the partial product in A. The carry from the addition is transferred to C.

The partial product in A is left unchanged if $Q = 0$. The P counter is

decremented by regardless of the value in Q. Registers A and Q are then shifted once to the right to obtain a new partial product.

$$AQ \leftarrow \text{shr } CAQ, C \leftarrow 0$$

CAQ is a composite register made up of flip-flop C and registers A and Q.

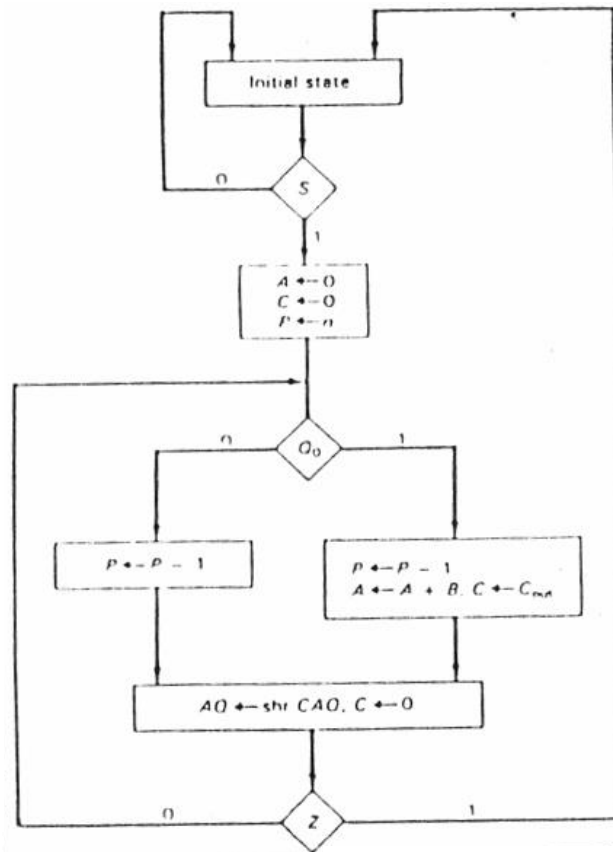


Fig.18 Flowchart for binary multiplier.

The value in the P counter is checked after each partial product. If the content of the P is not zero, status bit Z is equal to 0 and the process repeats to form a new partial product. The process stops when the P content reaches 0. The final product is available in A and Q, with A holding the most significant bits and Q the least. The following table represents the multiplication process.

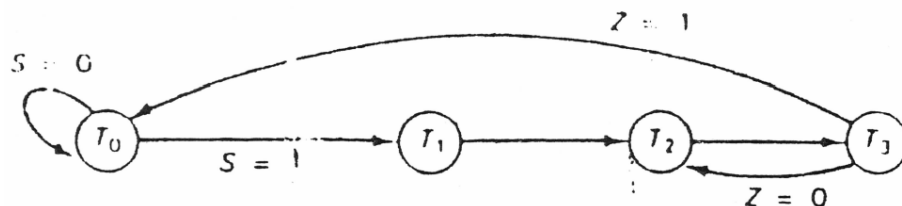
Numerical Example for binary multiplier:

Multiplicand B = 10111

	C	A	Q	P
Multiplier in Q	0	00000	10011	101
Q = 1 , add B		10111		
1st partial product	0	10111		100
SHR CAQ	0	01011	11001	
Q = 1 , add B			10111	
2nd	1	00010		011
SHR	0	10001	01100	
Q = 0 , SHR	0	01000	10110	010
Q = 0 , SHR	0	00100	01011	001
Q = 1		10111		
5th		11011		000
SHR		01101	10101	
Final product in AB is 01101 10101				

The design of a hardwired control is a sequential circuit problem. A flowchart is very similar to a state diagram. Although, it is possible to formulate the relationship between the flowchart and the state diagram, the conversion from one form to the other is not unique.

We start the design by assigning an initial state T_0 to the sequential controller. We then determine the required transmission to other states T_1 , T_2 , T_3 , and so on. Accordingly, from the previous flowchart, the control state diagram for the binary multiplier is shown.



State diagram for the binary multiplier.

T_0 : initial state.

T_1 : $A \leftarrow 0, C \leftarrow 0, P \leftarrow n$

T_2 : $P \leftarrow P-1$

$L = Q \cdot T_2$: $A \leftarrow A+B, C \leftarrow C_{out}$ $L =$ Control function

T_3 : $AQ \leftarrow \text{shr } CAQ, C \leftarrow 0$

Register transfer statements.

Remember that, all the operations are synchronized with the clock and are actually executed in response to a clock transition.

The block diagram of the control logic is shown in Fig.19. The inputs to the control logic are the external signal S and the two state conditions Z and Q_0 . The outputs are: T_0, T_1, T_2, T_3 and Q_0T_2 . The AND gate is shown separately although it is part of the control logic.

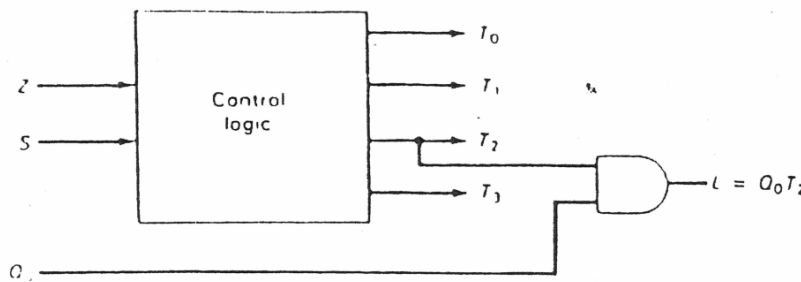


Fig.19 Control Block Diagram.

The outputs of the control circuit must be connected to the data processor subsystem to initiate the microoperations in the registers. Output T_1 must be connected to the load input of P and to the reset inputs of A and C . Output T_2 must be connected to the decrement input of P . Output L must be connected to the load input of A and C to receive the sum and output carry from the adder. Output T_3 must be connected to the shift control input of registers A and Q and to the reset input of C . Output T_0 has no effect (initial state).

Once the control sequence has been established; the sequential system must be designed. Since the control is a sequential circuit, it can be designed

by the sequential logic procedure as outlined before. However, in most cases this method is difficult to carry out because of the large number of states and inputs that a typical control circuit may have. As a consequence, it is necessary to use specialized methods for control logic design. Two such design procedures will be presented.

I. Sequence Register and Decoder

The sequence register and decoder method, as the name implies, uses a register to sequence the control states and a decoder to provide an output for each state. A register with n Flip-flops can have up to 2^n states and an n -to-2 line decoder will have up to 2^n outputs.

The control state diagram for the binary multiplier has four states and two inputs. Accordingly, we need two flip-flops for the register and a 2-to-4-line decoder. Although this is simple example, the procedure outlined below applies to more complicated situation as well.

The stable-table for the sequential control is shown. It is derived directly from the state diagram before.

State Table for control circuit

Present state		Inputs		Next state		Outputs			
G_1	G_0	S	Z	G_1	G_0	T_0	T_1	T_2	T_3
0	0	0	X	0	0	1	0	0	0
0	0	1	X	0	1	1	0	0	0
0	1	X	X	1	0	0	1	0	0
1	0	X	X	1	1	0	0	1	0
1	1	X	0	1	0	0	0	0	1
1	1	X	1	0	0	0	0	0	1

We designate the two flip-flops by G_1 & G_0 and assign the binary state 00, 01, 10, and 11 to T_0 , T_1 , T_2 , and T_3 respectively. Note that the input columns have do not care entries whenever the input variable is not used. The outputs of the sequential circuit are designated by variables T_0 through

T₃. The particular output variable that is equal to 1 at any given time is determined from the equivalent binary value of the present state. Thus, when the present state of the register is G₁G₀ = 00, output T₀ must be equal to 1 while the other three outputs remain 0. Since the outputs are a function of only the present state, they can be generated with a decoder circuit having 2-inputs G₁ & G₀ and four outputs T₀, T₁, T₂, and T₃.

Input equations:

The application of the classical method requires an excessive amount of work to obtain the simplified input equations for the flip-flops.

The design can be simplified if we take into consideration the fact that the decoder outputs are available for use in the design. The output T₀ through T₃ can be used to supply the present state conditions for the sequential circuit. From the next state conditions in the state table, we get:

$$\begin{aligned} D_{G1} &= T_1 + T_2 + T_3 \cdot \overline{Z} \\ \& \quad D_{G0} &= T_0 \cdot S + T_2 \end{aligned}$$

The logic diagram of the control circuit is shown in Fig.20. The output of the decoder are used to obtain the next state of the circuit according to the boolean input equations listed above. The outputs of the controller should be connected to the data processor part of the system to initiate the required microoperations.

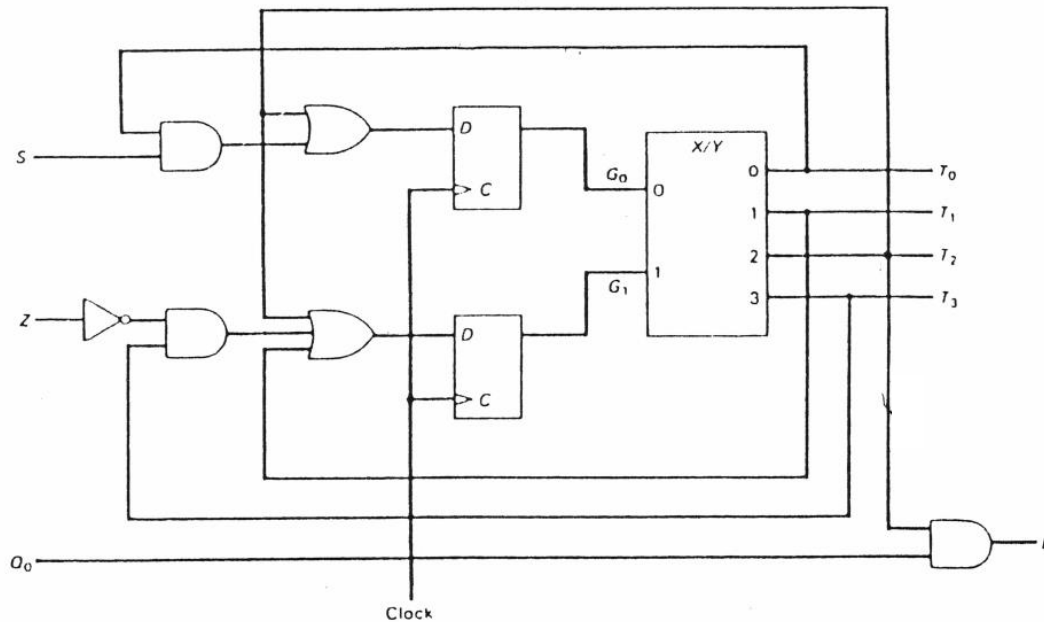


Fig.20 The logic diagram of the logic circuit.

II. One Flip-Flop per state

Another possible method of the control logic design is to use one Flip-Flop per state in the sequential circuit. Only one Flip-Flop is set to 1 at any particular time, all others are reset to 0. The single bit is made to propagate from one flip-flop to the other under the control of decision logic. In such configuration, each flip-flop represents a state that is activated only when the control bit is transferred to it.

We will demonstrate the design procedure by obtaining the control circuit specified by the state diagram before. Since there are four states in the state diagram, we choose four D flip-flops and label their outputs T_0 , T_1 , T_2 , and T_3 .

The input equation for setting each F.F to 1 is determined from the present state and the input conditions along the corresponding directed lines going into the state. For example, Flip-Flop T_0 is set to 1 with the next clock pulse transition if the circuit is in present state T_3 and input Z is equal to 1 or if the

circuit is in T_0 and $S=0$, i.e.

$$D_{T0} = T_0 \cdot \bar{S} + T_3 \cdot Z$$

where D_{T0} denotes the D input of Flip-Flop T_0 . Using this procedure for the other 3 flip-flops, we obtain the input equations:

$$\begin{aligned} D_{T1} &= T_0 \cdot S \\ D_{T2} &= T_1 + T_3 \cdot \bar{Z} \text{ and} \\ D_{T3} &= T_2 \end{aligned}$$

The logic diagram of the controller is shown in Fig.21. The outputs of the flip-flops must be connected to the corresponding inputs registers of the data processor subsystem. Initially T_0 must be set to 1 and all other flip-flops must be reset to 0 so that the flip-flop representing the initial state is enabled. Once started, the one flip-flop per state controller will propagate from one state to the other in the proper manner.

Sometimes, the one flip-flop per state controller is implemented with a register that has a common synchronous input for resetting all flip-flops initially to 0. In that case it is possible to transfer the circuit into initial state by modifying the input equation for T_0 as follows:

$$D_{T0} = T_0 \cdot \bar{S} + T_3 \cdot Z + \bar{T}_0 \cdot \bar{T}_1 \cdot \bar{T}_2 \cdot \bar{T}_3$$

The third term in the equation sets flip-flop T_0 to 1 with the first clock pulse, just after all flip-flops are reset to 0 asynchronously.

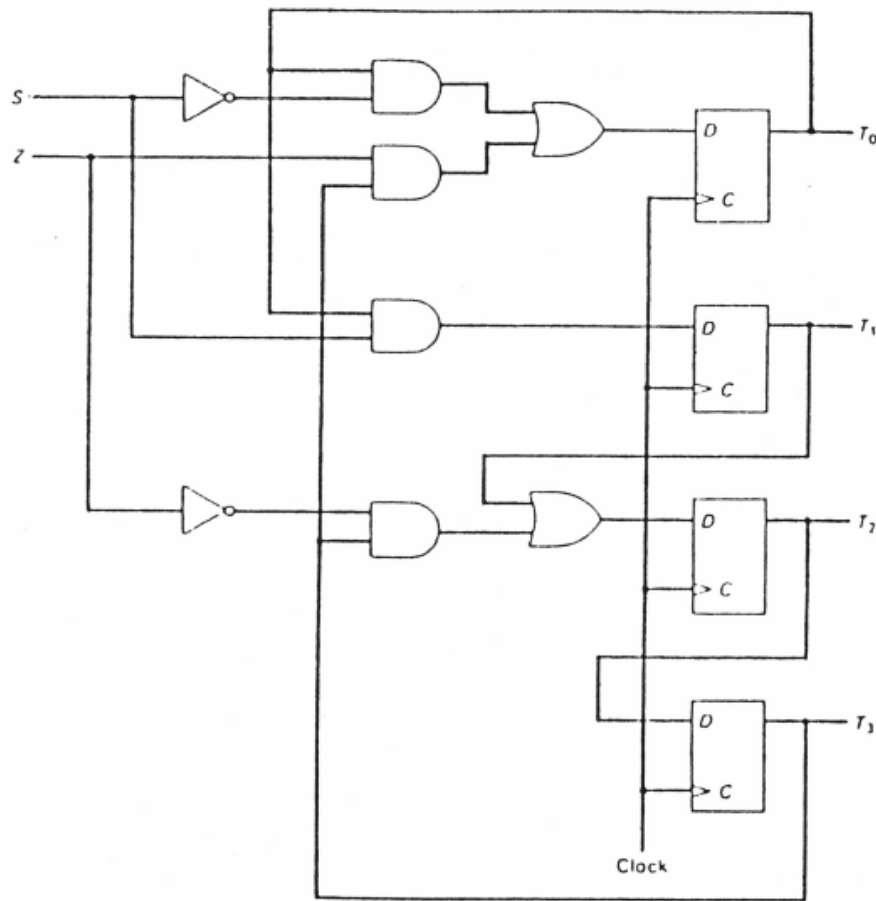


Fig.21 One flip-flop per state controller.

3.1.5 CPU control unit

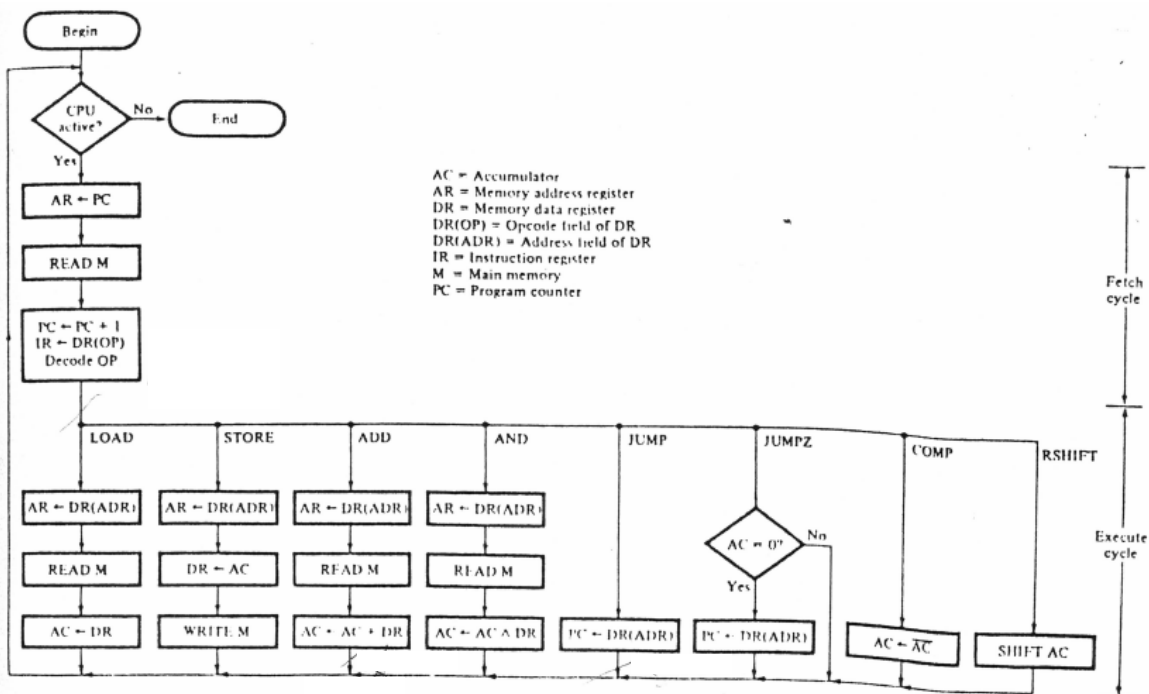
The design of a control logic for a CPU differs in degree but not in kind from the multiplier control unit of the preceding section. A CPU may contain several hundred control lines, which make control logic design quite complex. using the simple possible RISC-style CPU as an example , some of the design issues involved are examined.

Consider the hypothetical CPU, simple accumulator based CPU before. Assuming that it is required to execute the set of eight one address

instructions listed below. The algorithms needed to implement each instruction using the given hardware are easily derived.

Mnemonic	Description
LOAD X	$AC \leftarrow M(X)$ (transfer contents of memory location X to AC).
STORE X	$M(X) \leftarrow AC$
ADD X	$AC \leftarrow AC + M(X)$ (2's complement addition)
AND X	$AC \leftarrow AC \text{ (AND) } M(X)$ (Logical AND)
JMP X	$PC \leftarrow X$ (unconditional branch)
JMPZ X	if $AC=0$ then $PC \leftarrow X$ (conditional branch)
COMP	$AC \leftarrow AC$ (compl. AC)
RSHIFT	Rightshift accum.

The following flowchart describes the instruction fetch cycle common to all instructions, as well as the distinct execution cycle required for each of the specified instruction.



Operation of the eight-instruction CPU.

The microoperations in this flowchart determine the control signals and control points needed in the CPU. The following table lists a suitable set of control signals and their functions, while Fig. 21.a shows the appropriate positions of the corresponding control lines in the CPU.

Control signals of the simple CPU

Control signal	Operation controlled
C_0	$AC \leftarrow AC + DR$
C_1	$AC \leftarrow AC \text{ (AND) } DR$
C_2	$AC \leftarrow \overline{AC}$
C_3	$DR \leftarrow M(AR) \text{ (READ } M)$
C_4	$M(AR) \leftarrow DR \text{ (WRITE } M)$
C_5	$DR \leftarrow AC$
C_6	$AC \leftarrow DR$
C_7	$AR \leftarrow DR(ADR)$
C_8	$PC \leftarrow DR(ADR)$
C_9	$PC \leftarrow PC + 1$
C_{10}	$AR \leftarrow PC$
C_{11}	$IR \leftarrow DR(OP)$
C_{12}	RIGHT-SHIFT AC

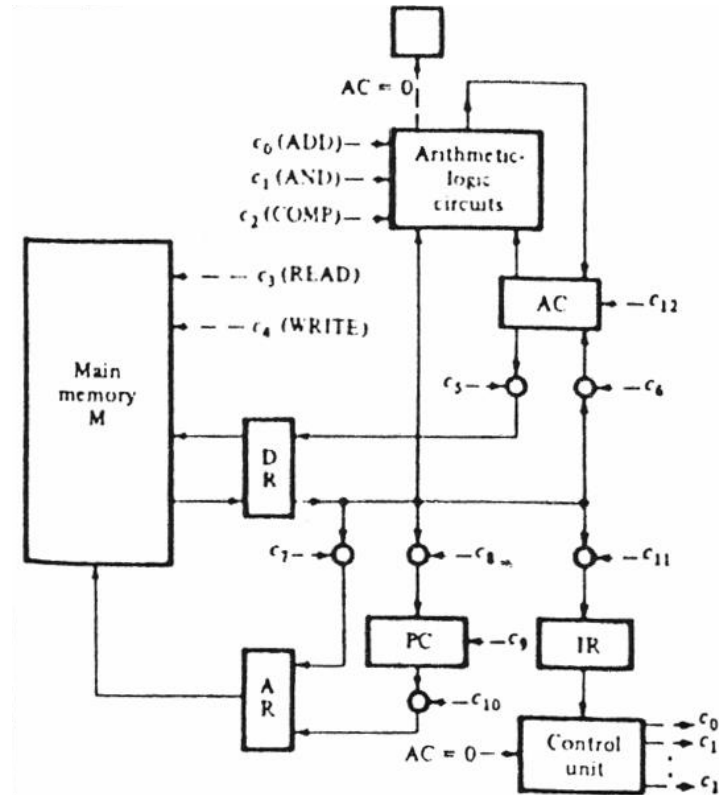


Fig. 21.a Structure of the simple CPU.

Considering that READ & WRITE M takes two time units each and every other microoperation takes one time unit. Inspection of the CPU chart reveals that 'slow' instruction such as ADD requires eight time units. The JMP requires only five units. We will therefore use a modulo-eight sequence counter driven by a clock whose period is equal to one time unit. The general structure of a simple hardwired control unit is shown in Fig.22.

In general, each control C_i can be defined by a logic equation of the form

$$C_i = \sum_j (\phi_j \sum_m I_m)$$

Where I_m is an output of the instruction decoder. In the case of an instruction requiring $j < k$ steps where k is the sequence counter modulus, the sequence counter may be reset after the j^{th} step.

For example, C_3 , which causes a memory read operation to take place, is defined by the following logic equation:

$$C_3 = \phi_2 + \phi_6 (LOAD + ADD + AND)$$

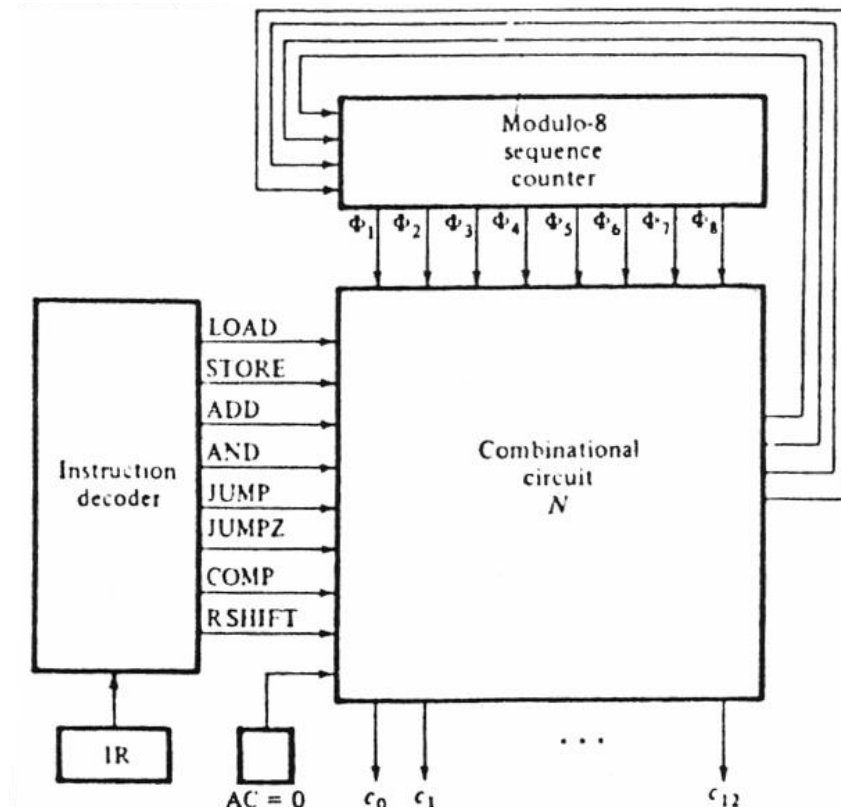


Fig.22 Hardwired CPU control unit using a sequence counter.

3.2 Microprogrammed Control

The purpose of the control unit in a digital system is to initiate a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a control word.

A control unit whose binary control variables are stored in a memory is called a microprogrammed control unit. Each word in control memory

contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. The control memory can be a read only memory (ROM).

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from the computer or from an auxiliary memory such as a magnetic disk. A memory which is part of control unit is referred to as a control memory.

The general configuration of a microprogrammed control unit is shown in Fig.23.

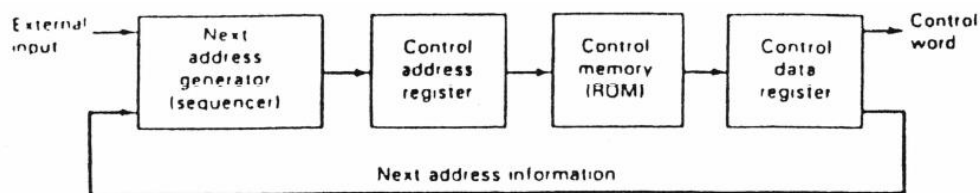


Fig.23 Microprogrammed control organization.

The microinstruction contains a control word that specifies one or more microinstructions for the data processor. The location of the next instruction may be the next one in sequence, or it may be located somewhere else in the control memory. For this reason, it is necessary to use some bits of the present microinstruction to control the generation of the address of the next instruction. The next address may also be a function of external input condition. Thus, a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory itself.

Typical functions of a microprogram sequencer are:

- i. Incrementing the control address register by 1.
- ii. Loading into the control address register an address from control memory.
- iii. Transferring an external address, and loading an initial address to start the control operations.

The control data register holds the present microinstruction while the

next address is computed and read from memory. The data register is sometimes called a pipeline register. This configuration requires a two phase clock, with one phase applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single phase clock to the address register. The control word and next address information are taken directly from the control memory.

3.3 Control of processor unit

To show the general properties of the microprogram organization, a configuration that include the control of a processor unit is included. A general purpose processor unit is shown in the following block diagram of Fig.24. It has seven registers $R_1 \rightarrow R_7$, ALU, a shifter, and four status bits C, Z, S, and V. A microoperation is selected with a control word of 16 bits, which is divided into 5 fields A, B, D, F, and H.

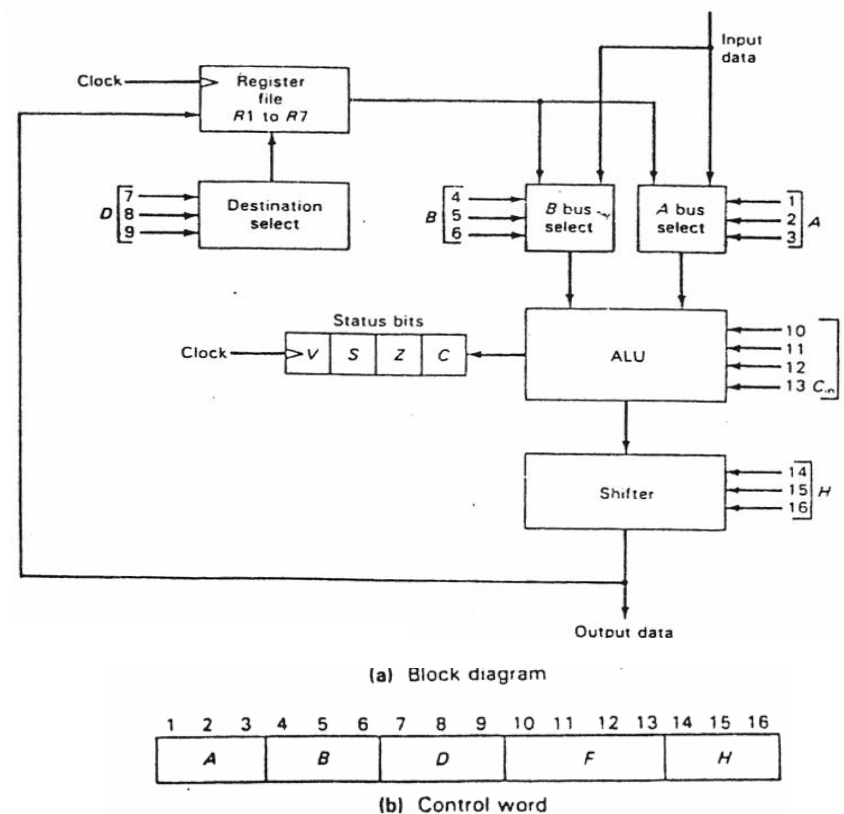


Fig.24 Processor unit with control variables.

A microprogrammed unit for controlling the processor is shown in Fig.25. It has a control memory ROM with 64 words of 26 bits each, a CAR, and MUX₁ & MUX₂. One bit selects between an external address and the address field of the microinstruction with MUX₁. Adding the 16 bits for the control word that selects a microoperation in the processor unit gives 26 bits for the microinstruction. Thus, each microinstruction read from control memory contains the 16 bits of a control word for the processor and 16 bits for selecting the next address in the control address register, CAR.

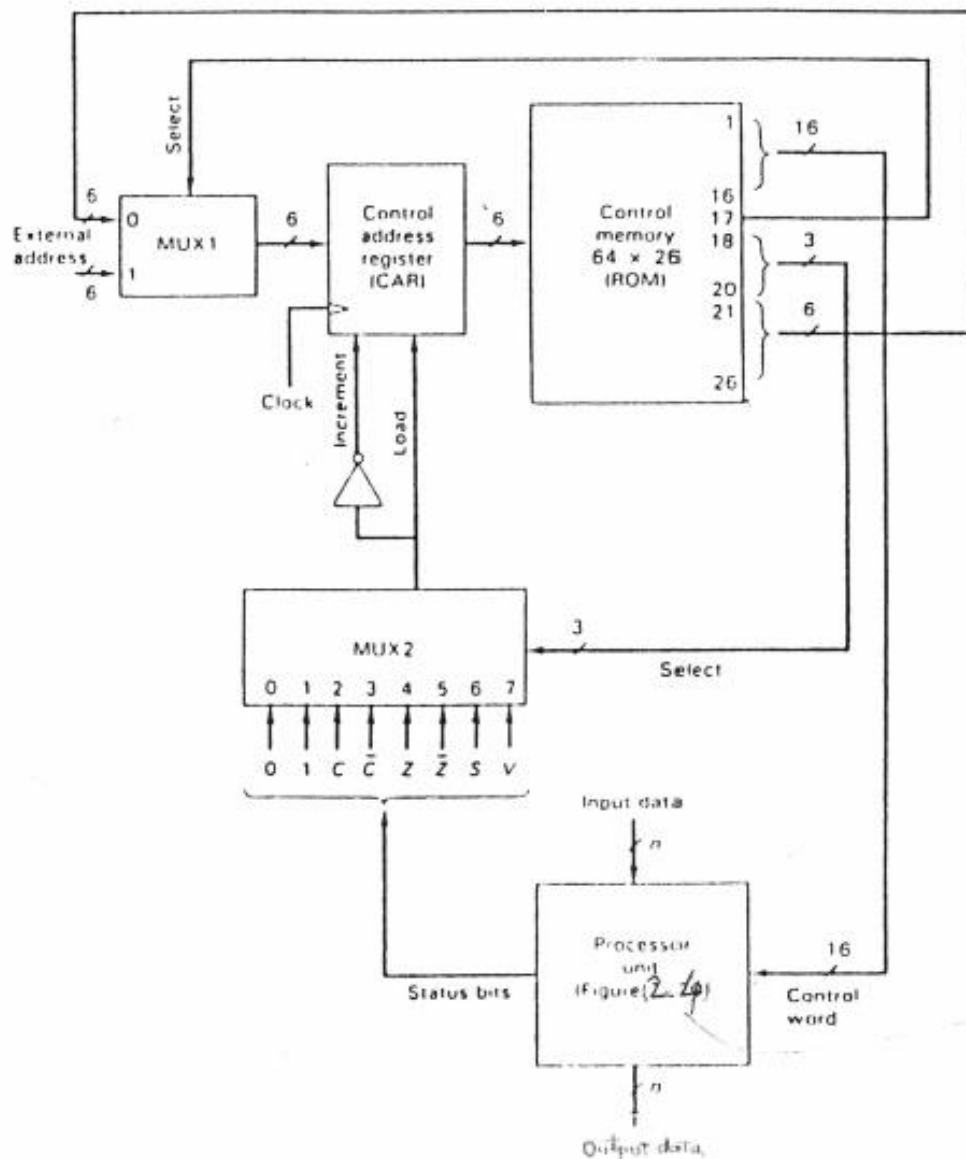


Fig.25 Microprogrammed control for processor unit.

3.3.1 Encoding of Microinstructions

The 26 bits of the microinstructions are divided into eight fields, as shown in the table.

Microinstruction format

Bits	Field	Symbol	Function
1- 3	A	INP, R1-R7	Right input of ALU
4- 6	B	INP, R1-R7	Left input of ALU
7- 9	D	NONE, R1-R7	Destination register
10-13	F	see table.1	ALU operation
14-16	H	see table.2	Shift operation
17	MUX1	INT, EXT	MUX1 sel. (int=0, ext=1)
18-20	MUX2	see table.3	MUX2 select
21-26	ADRS	Address no.	Address field

A microoperation is an elementary operation performed with the data stored in registers. In digital computers, there are four categories:

1. Register transfer microoperations.
2. Arithmetic microoperations.
3. Logic microoperations.
4. Shift microoperation

The ALU operations are listed in table 1. Each operation is recognized by the 4-bit binary code assigned to it and its three-letter symbol. The table also specifies how the status bits are affected by each operation. Z & S bits are affected by all the operations except TSF.

binary code	symbol	status bits				function
		Z	S	C	V	
0 0 0 0	TSF	N	N	N	N	Transfer A
0 0 0 1	INC	Y	Y	N	N	Incr. A by 1
0 0 1 0	ADD	Y	Y	Y	Y	Add A + B
0 1 0 1	SUB	Y	Y	Y	Y	Sub A - B
0 1 1 0	DEC	Y	Y	N	N	Decr. A by 1
0 1 1 1	TRC	Y	Y	0	N	Transf. A & reset CAR
1 0 0 0	AND	Y	Y	N	N	A AND B
1 0 1 0	OR	Y	Y	N	N	A OR B
1 1 0 0	X-OR	Y	Y	N	N	A XOR B
1 1 1 0	COM	Y	Y	N	N	Complement A.

Table 1 ALU operations (F field)

N : Status bits are not affected.

Y : Status bits are affected by the operations.

0 : Reset to 0.

The shifter operations are listed in table 2. SHL & SHR receive Zero in their corresponding serial input. ROL & ROR rotate the bit from the serial output into the corresponding serial input. RLC & RRC insert the value of the status bit C into the serial input and transfer the serial output bit into the C flip-flop.

Binary code	Symbol	Function
0 0 0	NSH	No shift.
0 0 1	SHL	Shift left with ser.inp=0
0 1 0	SHR	Shift right with ser.inp=0
0 1 1	ZERO	All zero's in outp. of shifter.
1 0 0	RLC	Rotate left with carry.
1 0 1	ROL	Rotate left.
1 1 0	ROR	Rotate right.
1 1 1	RRC	Rotate right with carry.

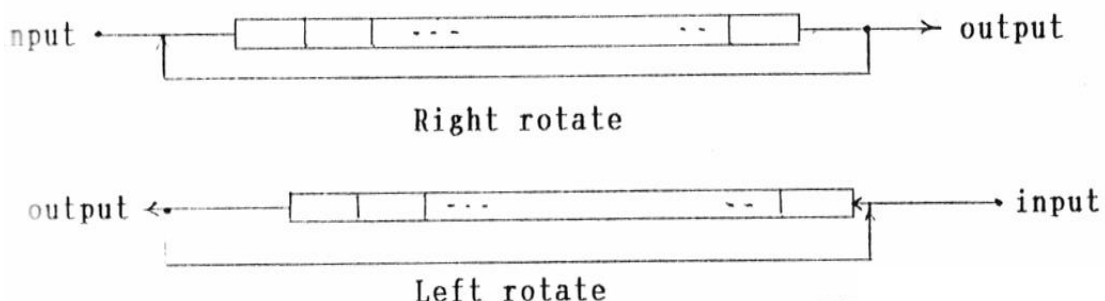
Table 2 Shifter operations.

The binary select input to MUX2 and a symbolic name for each combination are listed in table 3.

The code 000 chooses a 0 input for the MUX2 which causes CAR to be incremented.

Binary code	Symbol	Function
0 0 0	NEXT	Go to next address by incr. CAR
0 0 1	LAD	Load add. into CAR (branch uncond.)
0 1 0	LC	Load on carry (branch if C=1)
0 1 1	LON	Load on nocarry (branch if C=0)
1 0 0	LZ	Load on zero (branch if Z=1)
1 0 1	LNZ	Load on nonzero (branch if Z=0)
1 1 0	LS	Load on sign (branch if S=1)
1 1 1	LV	Load on overflow (branch if V=1)

Table 3 Select input to MUX2.



statement:

$$R_1 \leftarrow R_1 \text{ (AND) } R_2, \text{ CAR} \leftarrow \text{CAR} + 1$$

The binary values for the microinstruction are listed under the symbols in each field. The fields under the dash are not used, and therefore any binary number can be used. For convenience, and consistency, these places are marked with 0's. The 26-bit words that stored in control memory at address 36 is:

00101000110000000000000000.

2. The equivalent register transfer statement in address 40 is:

$$R_3 \leftarrow R_3 - 1, \text{ CAR} \leftarrow 43$$

It decrements R_3 and branches to the microinstruction at address 43. The branching to address 43 is specified by the LAD and INT symbols in MUX_2 and MUX_1 fields.

Microprogram Example

The number of 1's stored in R_1 is to be counted. Register R_2 is set to that number. For example, if $R_1 = 00110110$ the microprogram routine counts the four 1's in the register and sets R_2 to the binary number 100. The count is done by shifting each bit from R_1 at a time into the C states bit and incrementing R_2 each time that C is equal to 1.

The flowchart for the routine is shown in Fig.26.

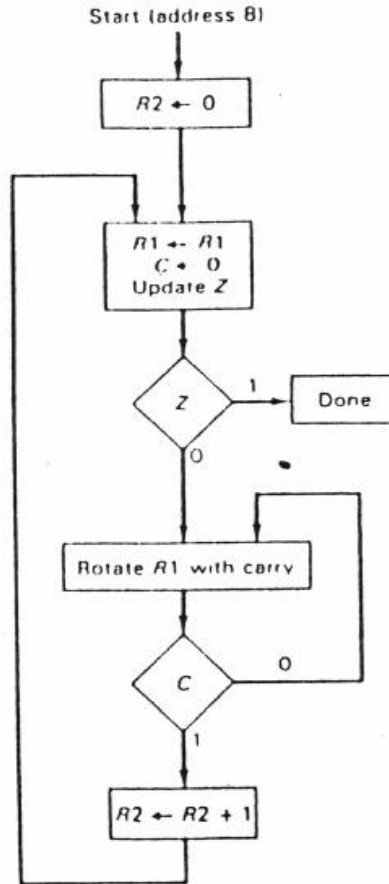


Fig.26 Flowchart for counting the number of 1's.

The routine starts for example, from 8. Register R_2 is initially reset to 0. The content of R_1 is transferred through the ALU to update the Z-bit and reset the carry bit C to 0.

If $Z=1$, it means that the content of R_1 is zero, which signifies that there are no ones in the number. In that case, the microprogram routine terminates with R_2 equal to 0. If $Z=0$, contents of $R_1 \neq 0$, and therefore, there are some ones stored in it.

R_1 together with the carry are shifted by rotation as many times as necessary until a 1 is transferred into C. Register R_2 is incremented by 1 for every 1 detected in C, and the microprogram branches back to check if R_1 is equal to 0. The program loop is repeated until all the 1's in R_1 are counted. The microprogram in symbolic form is listed in the following table.

CAR address	Control word					select		Address field
	A	B	D	F	H	MUX1	MUX2	
8	--	--	R2	TSF	ZERO	---	NEXT	----
9	R1	--	R1	TRC	NSH	---	NEXT	----
10	--	--	NONE	TSF	NSH	EXT	LZ	----
11	R1	--	R1	TSF	RRC	---	NEXT	----
12	--	--	NONE	TSF	NSH	INT	LNC	11
13	R2	--	R2	INC	NSH	INT	LAD	9

Microprogram counting the number of 1's.

The routine starts at address 8 by cleaning R_2 to 0. At 9 the ALU operation TRC transfers contents of R_1 to ALU, resets the carry to 0, and update the Z and S bits.

At address 10, checks Z and if $Z=1$, R_1 contains all 0's and the routine is terminated, by accepting a new external address. If $Z=1$, it continues to 11. RRC places the least significant bit of R_1 into C and shifts R_1 to the right. C is checked at 12. If $C=0$, control goes back to 11 to rotate again. When $C=1$, it goes to 13 to increment R_2 and then returns to 9, and so on.

The microprogram concept is a systematic procedure for designing the control logic of a digital system. Once the microprogram, format is established, the design is done by writing a microprogram, which is similar to writing a program for a computer. For this reason, the microprogram method is sometimes referred to as firmware to distinguish it from the purely hardware method (which is also called hardware control) and the software concept which constitutes a purely programming method.

Problems

[1] Define the following terms in your own words:

- a. data processor subsystem.
- b. control subsystem.
- c. hardwired control.
- d. microprogrammed control.
- e. control memory.
- f. control word.
- g. microinstruction.
- h. microprogram.
- i. microoperation.

[2] List the contents of registers A,Q,P,and C during the process of multiplying the two unsigned binary numbers 11111 (multiplicand)and 10101 (multiplier).

[3] The control state diagram of the binary multiplier in the lectures, does not use variable Q_0 as a condition for state transition. Redesign the control for that multiplier so that Q_0 appears as a condition in the state diagram and as an input in the state table.

[4] Give a logic diagram (at a reasonable level) for the control section of an ALU. The ALU is capable of the following operation:

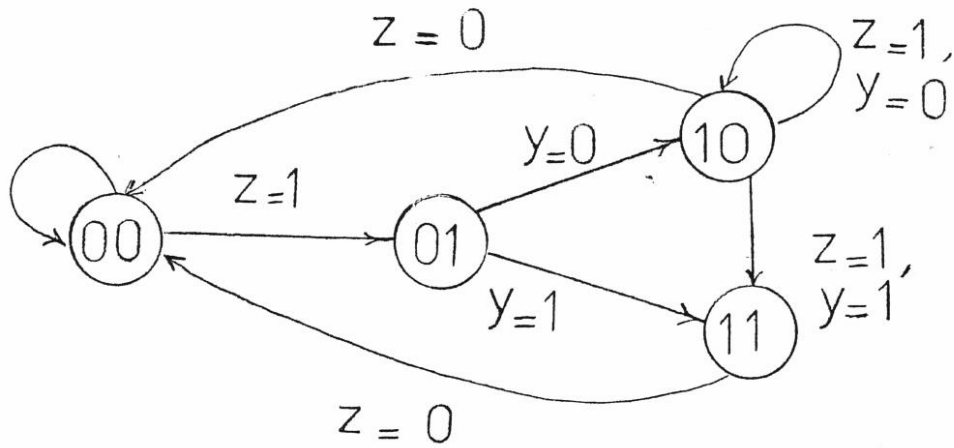
C_1	C_2	Function
0	0	$F = A + B$
0	1	$F = A \text{ NAND } B$
1	0	$F = \text{NOT } A$
1	1	$F = A \text{ OR } B$

[5] Design a digital system with three 16-bit registers AR, BR, and CR to perform the following operations:

- a. Transfer two 16-bit signed numbers to AR and BR after a start signal S is enabled.
- b. If the number in AR is negative, divide the contents of AR by two and transfer the result to register CR.

[6] The state diagram of a control logic is shown. It has four states and two

inputs Y & Z. Design the control by the sequence register and decoder method.



Problems

[1] A microinstruction stored in address 35 in the control memory of the processor unit, performs the operation:

$$R_1 \leftarrow R_1 + R_2, \text{ CAR} \leftarrow \text{CAR} + 1$$

Give the microinstruction in symbolic form using the symbol LAD and MUX₂.

[2] List the symbol and binary microinstruction for the following register transfer instructions.

- a. $R_3 \leftarrow R_1 - R_2, \text{ CAR} \leftarrow 17$
- b. $R_5 \leftarrow \text{shr}(R_4 + R_5), \text{ CAR} \leftarrow \text{CAR} + 1$
- c. $R_6 \leftarrow R_6, C \leftarrow 0, \text{ CAR} \leftarrow \text{CAR} + 1$

[3] A certain processor has a microinstruction format containing 10 separate control fields $C_0 : C_9$. Each C_i can activate any of n_i distinct control lines, where n_i is specified as follows:

$$i = 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9$$

$$n_i = 4 \ 4 \ 3 \ 11 \ 9 \ 16 \ 7 \ 1 \ 8 \ 22$$

What is the minimum number of control bits needed to represent the 10 control fields?

What is the maximum number of control bits needed if a purely horizontal format is used for all the control information?

[4] Draw a logic diagram showing how to construct a microprogram sequencer for:

- a. 64 X 12-bit control memory.
- b. 12 X 64-bit control memory.

[5] Write microprogram (starting from address 1) that checks the sign of the binary number stored in register R_1 . If the number is positive, it is divided by two. If it is negative, it is multiplied by two. If an overflow occurs after multiplication, R_1 is reset to 0.

[6] Write a microprogram that compares two unsigned binary numbers stored in register R_1 and R_2 . The register containing the smaller number is then reset to 0. If the two numbers are equal, both registers are reset to 0.

CHAPTER FOUR

MEMORY SYSTEMS

One of the most basic function of a computer is the retrieval of information stored in a memory element. This action is needed to obtain the instruction to perform it is also needed to obtain the data on which the instruction operates. Every computer system contains a variety of devices to store the instructions and data required for its operation. These storage devices plus the algorithms (either implemented by hardware or software) needed to control or manage the stored information constitute the memory system of the computer.

In general, it is desirable that processors should have immediate and uninterrupted access to memory, so the time required to transfer information operate at, or close to , its maximum speed.

A simple representation of a memory that can be accessed by a processor and I/O devices is shown in Fig.27.

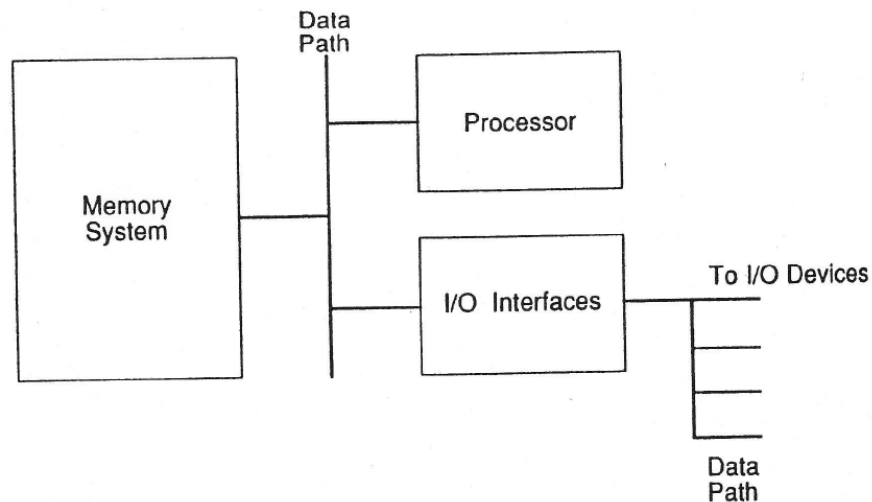
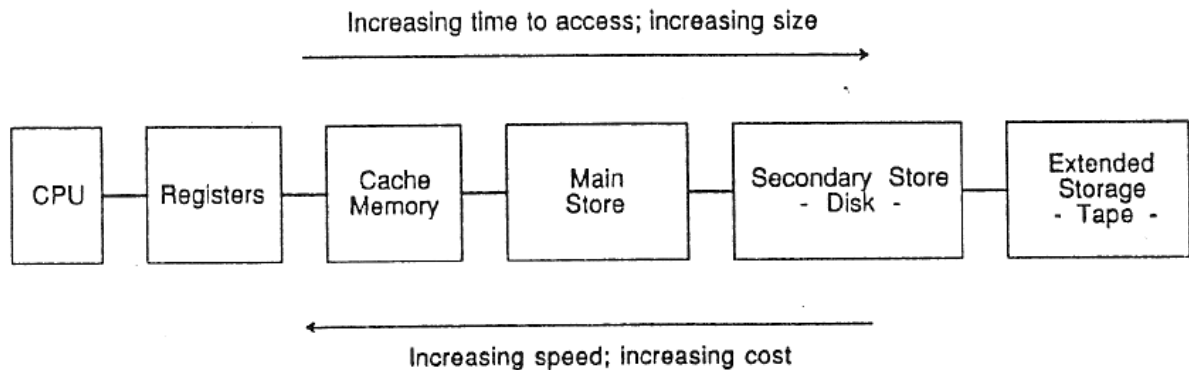


Fig.27 Memory system and connections to processor and I/O interfaces.

4.1 Memory Hierarchy

Hierarchy of memories should lead to sufficient storage for the problems to be solved. Therefore, the designers looked forward to the constructing of a hierarchy of memories, each of which has greater capacity



than the preceding, but which is less quickly accessible. The machine of today indeed match this concept, and can be represented by the following diagram shown in Fig.28.

Fig.28 Block diagram of Hierarchical memory system.

The fastest memory elements are those closest to the processor, most systems have a small number of very high speed locations, which we call a register bank. The access time T for the registers is minimal, and in general the information stored in registers is available in the same cycle as it is needed. Because of the high cost of this type of storage, the amount of register available in a system is relatively small (from 8 to 16 registers in most general purpose systems, to over a 100 in some special purpose and RISC systems).

The next element is often a cache memory. The purpose of a cache is to enhance the operating speed of the processor. The T of a cache is on the order of 2 CPU cycle times. The amount of memory available here is generally small in comparison to the other elements of the system. For example, the VAX 11/780 has a 2-Kbyte, and some other processors have even smaller caches. Many newer systems have caches that contain 16 Kbytes to 64 Kbytes or more incorporated with the processing unit.

The purpose of the cache memory is to maintain current information for rapid retrieval. This is done in a manner that is transparent to the user. The programmer does not know of the existence of the cache, except that the speed of the system is enhanced over a system with no cache memory. The management of the data is done in a fashion determined at design time, in a contrast to the virtual memory.

The information in a cache memory is a high speed copy of what is in main store, which is the standard memory of the computer. The amount of storage in the main store is system dependent. The technology is now (1990) such that it is possible to get 8 Mbytes in eight packages. The T_a of the main store is about an order of magnitude greater than the T_a for cache. The information resident in main store for a standard computer system is a sufficient amount of the operating system to maintain a continuity of action. In addition, the active portions of the user programs and data sets are available as well.

The portions of the operating system and user programs and data that are not active are kept on the next level of the hierarchy, the secondary store. The purpose of the secondary store of this hierarchy is to maintain copies of all of the programs and data needed by the computer.

Generally, this will be a disk, although it could be any block-oriented storage device with a large capacity. Such devices have been built with CCD's, bubble memories and large RAM's. Note that the cache is created from a relatively small amount of high speed RAM, and main store is also electrically and randomly accessible, but with lower cost, slower devices than the cache.

The ratio for $\{T_a \text{ (main store)} / T_a \text{ (cache)}\}$ is of the order of 10, but the ratio for $T_a \text{ (secondary store)} / T_a \text{ (main store)}$ is on the order of 100,000. The last member of the hierarchy is the extended storage. This consists of information stored on magnetic tape, which is slow in comparison to the disk storage. This storage is generally used for permanent storage of programs and data as well as transfer of information from computer to another.

4.2 Standard Memory Systems

The storage of information in computer systems is accomplished by utilizing collections of individual storage elements, each of which is capable of maintaining a single bit. Thus, for a device to be useful as a memory element it must have two stable states, a reliable mechanism for setting the device to one state or the other, and a mechanism for interrogating the state. Semiconductor memories may have static memory cells or dynamic memory cells. Storage cells in a RAM are fabricated with either bipolar or MOS components. The TTL cell is operated statically, whereas some MOS units

are available which operate in a static mode and others in a dynamic mode. A static MOS RAM cell with six transistors is shown in Fig.29. Also, Fig. 30 shows a one-MOSFET dynamic memory.

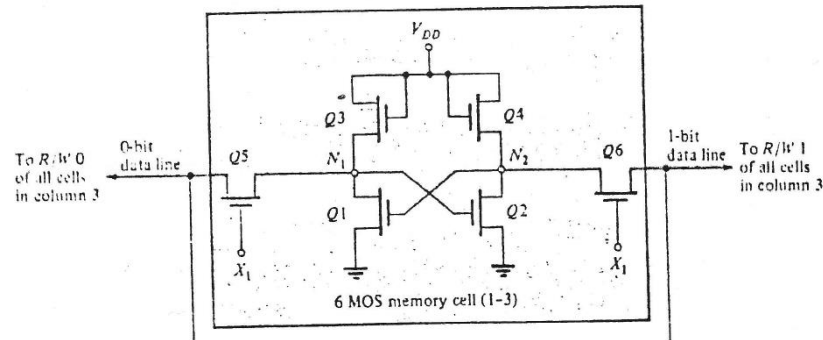


Fig.29 A static MOS RAM cell with 6 n channel MOSFETs.

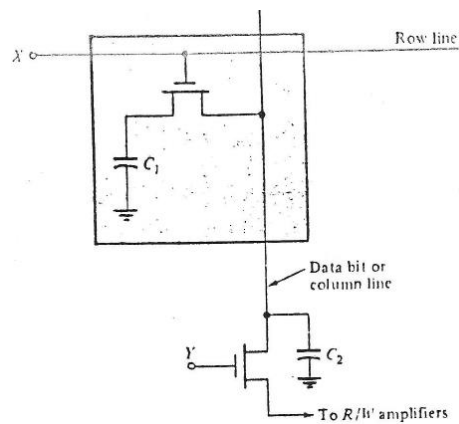


Fig.30 A one-MOSFET dynamic memory.

Another dynamic RAM cell is shown in Fig.31.

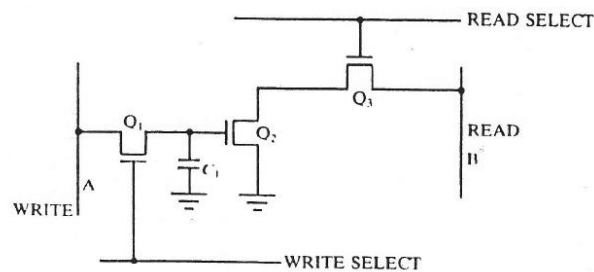


Fig.31 Dynamic RAM cell.

The storage of information in the cells is only a part of the memory problem. The bits stored must be organized in a reasonable fashion to access the information. The two most prevalent mechanisms are random access and serial access. Examples of serial access devices include magnetic surface systems, (such as tape and disk), and serial semiconductor systems (such as shift registers and CCD's).

The mechanism used to decode the address may be one dimensional (1-D) and two dimensional (2-D) decoding schemes. The (1-D) scheme accepts N-bit address and uses N-2 decoder, while the (2-D) scheme accepts N-bit address and divides it into two groups X & Y where $X + Y = N$, these two groups of address lines control X to 2^X & Y to 2^Y decoders.

Fig.32 shows 1-D and 2-D memory arrays.

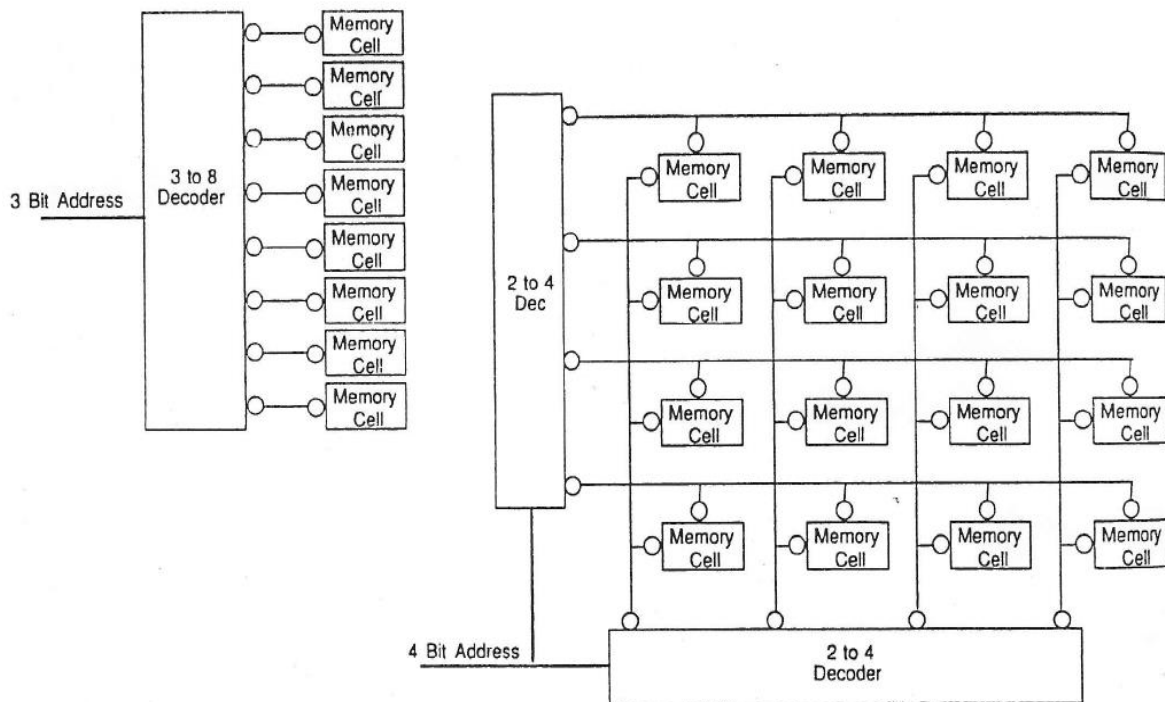


Fig.32 Memory cells organized in 1-D and 2-D memory arrays.

4.3 Cache Memory (Speed-up for main store)

Cache memories are (relatively) small, high speed memories inserted into the system between the processor and the main store. The purpose of the cache memory is to speed up the processing rate by allowing the processor to execute at a higher rate than the possible by using the main store alone. It utilizes many of the same concepts used with virtual memories, but in a slightly different fashion.

The basic operation of the cache is as follows: when the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word.

A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred so that future references to memory may find the required word in the fast cache.

The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If it is not found in cache, then it is in the main memory and it counts as a miss. The hit ratio is (no. of hits)/ (no. of hits + miss). The basic characteristic of cache memory is its fastest access time.

The ratio of access speed $R = (T_{A \text{ main store}} / T_{A \text{ cache}})$. Access times for main store and cache memories improve each year, typical times (1990) might be 250 nsec for $T_{A \text{ main store}}$ and 40 nsec for $T_{A \text{ cache}}$, ($R_{CA} = 250/4 = 6.25$).

The effective access time of the memory system, considering the cache and main store memories together as a single system T is given by:

$$T_{\text{eff}} = T_{CA} + (1-h) T_{MS}$$

Where h is the hit rate.

Fig.33 shows an example of cache memory. The speed up of the system S is:

$$S = T_{MS} / T_{eff}$$

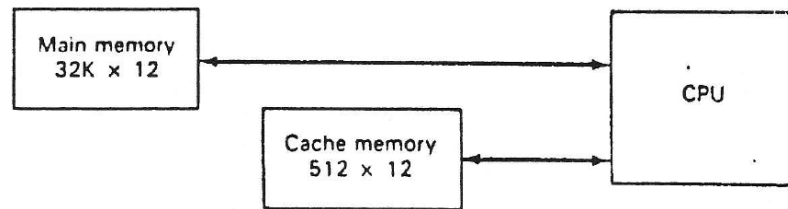


Fig.33 Example of cache memory.

The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest in considering the organization of cache memory.

1. Associative Mapping:

The fastest and the most flexible cache organization uses an associative memory. This organization is illustrated in the following table:

Argument register	
Address	Data
0 1 0 0 0	3 4 5 0
0 2 7 5 6	5 4 2 7
2 3 4 8 7	4 3 5 8

Associative Mapping cache (All numbers are in octal).

The associative memory stores both the address and content (data) of the memory word.

This permits any location in cache to store any word from main memory. The above table shows three words stored in cache. The address value of 15-bits is shown as a 5-digits octal number and its corresponding 12-bits data word is shown as a 4-digit octal number. A CPU address of 15-bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bits data is read and sent to the CPU.

2. Direct Mapping

Associative memories are expensive compared to random access memories because of the added logic associated with each cell.

Fig.34 shows the addressing relationships between main memory and cache memory.

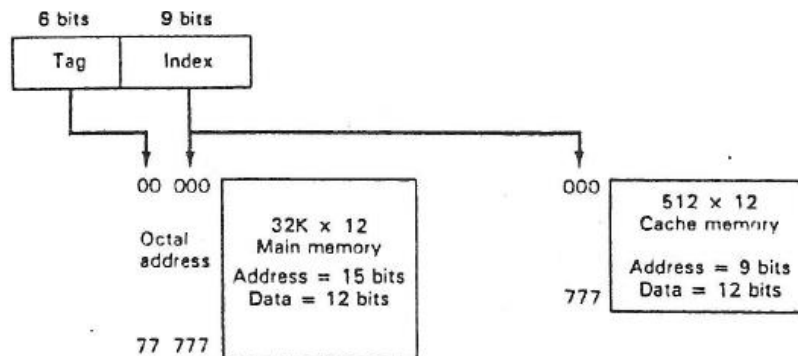


Fig.34 Addressing relationships between main memory and cache memory.

In the general case, there are 2^k words in cache memory and 2^n in main memory. The n bit memory address is divided into two fields: k -bits for the index field and $(n-k)$ for the tag field.

The direct mapping cache organization uses the n -bit address to access the main memory and the k -bit address to access cache. Each word in the cache consists of the data word and its associated tag.

When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the

index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag field in the word read from cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match there is a miss and the required word is read from main memory.

Fig.35 shows a direct mapping cache organization.

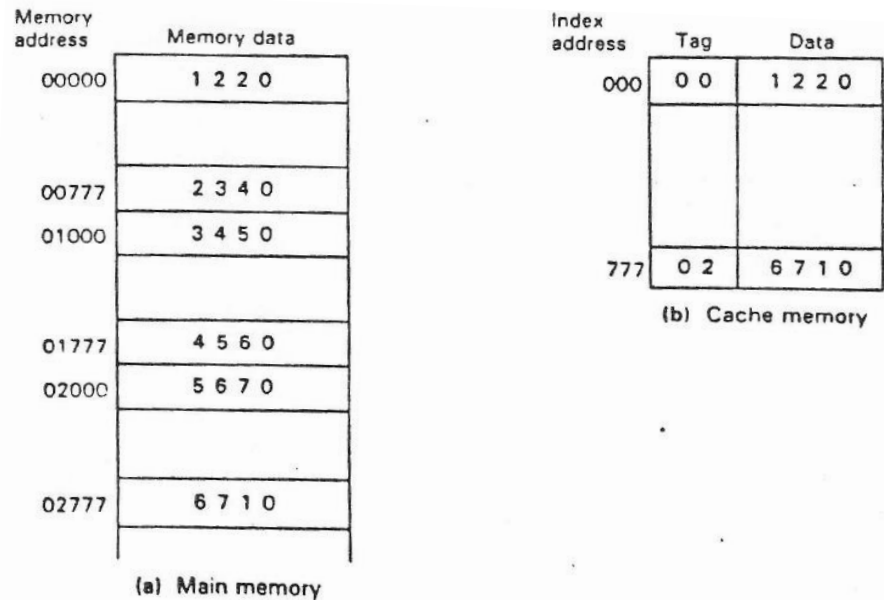


Fig.35 Direct Mapping Cache Organization.

It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio may drop considerably if two or more words with addresses having the same index but different tags are accessed repeatedly.

Direct Mapping Operation:

Consider the above numerical example. The word at address zero is presently stored in cache (index = 000, tag = 00, data = 1220). Suppose that the CPU wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is

transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

3. Set-Associative Mapping

A third type of cache organization, called set associative mapping is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address. A set associative mapping cache with set size of two is represented in the following table:

Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

Set Associative Mapping cache with set size of two.

The words stored at addresses 01000 and 02000 of the main memory before are stored in cache memory at index address 000.

Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU is then compared with both tags in the cache to determine if a match occurs. The hit ratio will improve as the set size increases.

4.4 Associative Memory

Many data processing applications require the search of items in a table stored in memory. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a matching occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm.

The time required to find an item stored in a memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an associative memory or content addressable memory.

An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

4.4.1 Hardware organization

Fig.36 shows a block diagram of an associative memory.

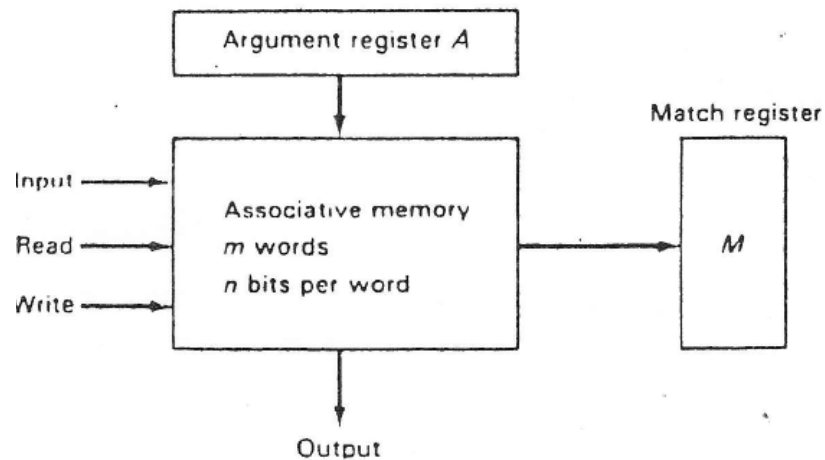


Fig.36 A Block diagram of associative memory.

It consists of a memory array and logic for m words of n -bit/word. The argument register A has n -bits, one for each bit of a word.

The match register has m bits, one for each word in memory. Each word is compared in parallel with the content of the argument register. The word that match the bits of the argument register set a corresponding bit in the match register. Reading is accomplished by a sequential access to memory for those words that have corresponding bits in the match register that have been set.

Fig.37 shows a match logic for two words of associative memory.

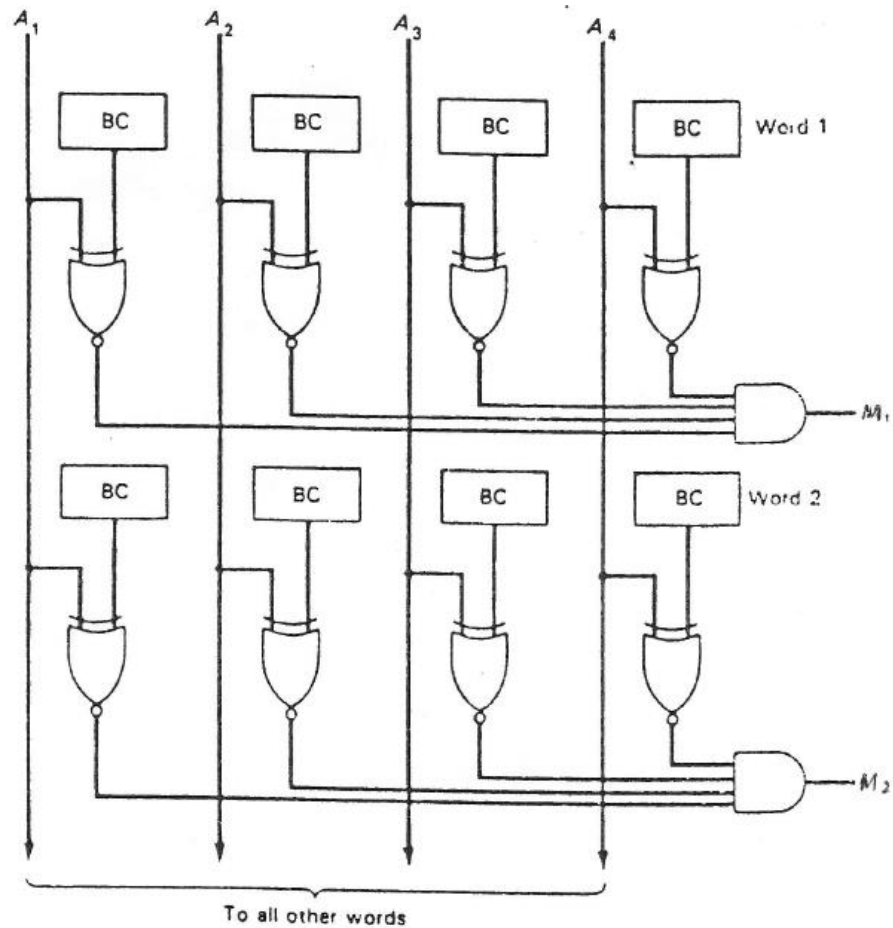


Fig.37 A match logic for two words of associative memory.

A_1 through A_m are the bits from the argument register that must be compared. M_j are the match signal $j=1,2,3,\dots,m$ where m is the number of words in the memory.

4.4.2 Read and Write operations

The matched words are read in sequence by applying a read signal to each word whose corresponding M bit is 1. In most applications, including application in memory management hardware, the associative memory stores a table with no two identical items.

By connecting output M_j directly to the read line of the same word, the content of the matched word will be presented automatically at the output of the memory.

An associative memory must have a write capability for storing the information to be searched. Since unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. The register, sometimes called a tag register must have as many bits as there are words in memory.

For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by setting its tag to be 0. Words are stored in memory scanning the tag register until the first 0 bit is encountered. This gives the first available in-active word and a position for writing a new word.

4.5 Virtual Memory

In memory hierarchy systems, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory, as they are needed by the CPU. Virtual memory is a concept used in some computer systems that permits the user to construct his program as though he had a large memory space, equal to the total of auxiliary memory. There are three main reasons for using virtual memory:

1. To free programmers from the need to carry out storage allocation and to permit efficient sharing of memory space among different users.
2. To make programs independent of the configuration and capacity of the memory systems used during their execution.
3. To achieve the high access rates and low cost per bit that is possible with a memory hierarchy.

NOTE: Virtual memory implies the use of a more economical storage medium, which has enough capacity for all programs and data. This medium is usually a disk.

Each address that is referenced by the CPU goes through an address mapping from the so-called virtual or logical address to a physical address in

main memory $f: L \rightarrow P$. A virtual memory system provides a mechanism for translating program generated address into corresponding main memory locations. The translation or mapping is handled automatically by the hardware by means of mapping tables.

In a multiprogramming environment where many programs reside in memory, it becomes necessary to move programs and data around in memory, to vary the amount of memory in use by a given program, and to prevent a program from changing other programs. The demands on computer memory brought about by multiprogramming have created the need for a memory management system. A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory.

4.5.1 Address Space and Memory Space

The address used by a program is called a virtual or logical address, and the set of such addresses is referred to as the address space. An address in main memory is called a location or physical address.

The set of such location is called the memory space. The address space is allowed to be larger than the memory space in computers with virtual memory.

The address mapping from address space to physical space can be simplified if the addresses are divided into groups of fixed size. The logical memory is broken down into groups of equal size called pages.

Memory management systems divide programs and data into logical parts called segments. A segment may be generated by the programmer or by the operating system. Examples of segments are: subroutines, arrays of data, tables of items, or user's programs.

The term block or page frame refers to groups of equal size in physical memory. Although, both a page and a block are split into groups of 1K, or something other, words, a page refers to the organization of address space, while a block refers to the organization of memory space.

Problems

[1] The 1DT7164 is an 8K*8 static RAM with 13 address lines and 8 input/output lines, as well as two chip selects, a write enable and output enable. Using this device, design a 64-Kbyte memory using 1-D organizational techniques.

[2] A certain cache memory machine uses a cache that can store 512 blocks of 64 words each. Assume a main memory size of 1.048.576 words. Which bits of the word address should specify the block number?

[3] Obtain the complement function for the match logic of one word in an associative memory. In other words, show that the complement of M is obtained from the logical sum of Exclusive-OR functions. Draw the logic diagram for M and terminate it with an inverter to obtain M.

[4] A digital computer has a memory unit of 64K*16 and a cache memory of 1K words. The cache uses direct mapping. How many bits are there in the tag and index fields of the address?

[5] The access time of a cache memory is 100 nsec and that of main memory 1000 nsec. The hit ratio is 0.9. What is the average access time of the system?

[6] An address space is specified by 24 bits and the corresponding memory space is 16 bits.

- a. How many words in the address space?
- b. How many words in the memory space?
- c. If a page consists of 2K words, how many pages and blocks are there in the system?

CHAPTER FIVE

REGISTER TRANSFER

AND

COMPUTER

OPERATIONS

5.1 Register Transfer

The register transfer operations of digital systems are specified by the following three basic components:

1. The set of registers in the system and their function.
2. The operations that are performed with the information stored in the registers.
3. The control that supervises the sequence of operations in the system.

The operations performed on the information stored in registers are called microoperations. A microoperation is an elementary operation that can be performed in parallel on a string of bits during one clock pulse. The result of the operation may replace the previous binary information in a register or may be transferred to another register.

The registers in a digital system are designated by capital letters (sometimes followed by numerals) that denote the function of the register. For example, the register that holds an address or the memory unit is usually called the memory address register AR. Other designations for the registers are PC, IR, R_1 , and R_2 . The individual flip-flops in an n -bit register are numbered in sequence from zero through $n-1$, starting from 0 in the right most position.

Fig.38 shows the representation of registers in block diagram form.

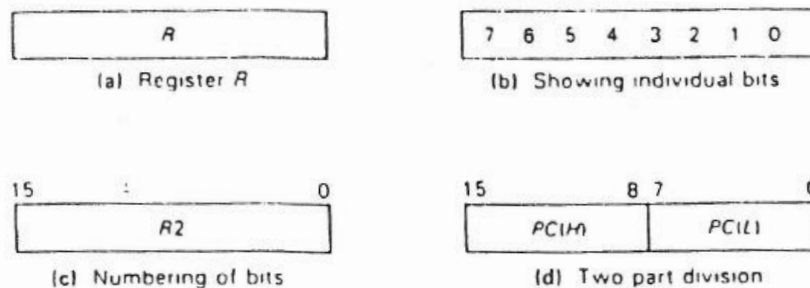


Fig.38 A representation of registers in block diagram form.

The statement $R_2 \leftarrow R_1$ denotes a transfer of the contents of R_1 into R_2 . It designates a replacement of the contents of R_2 by the contents of R_1 . By definition, the contents of the source register R_1 do not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register and that the destination register has a parallel load capability as shown in Fig.39.

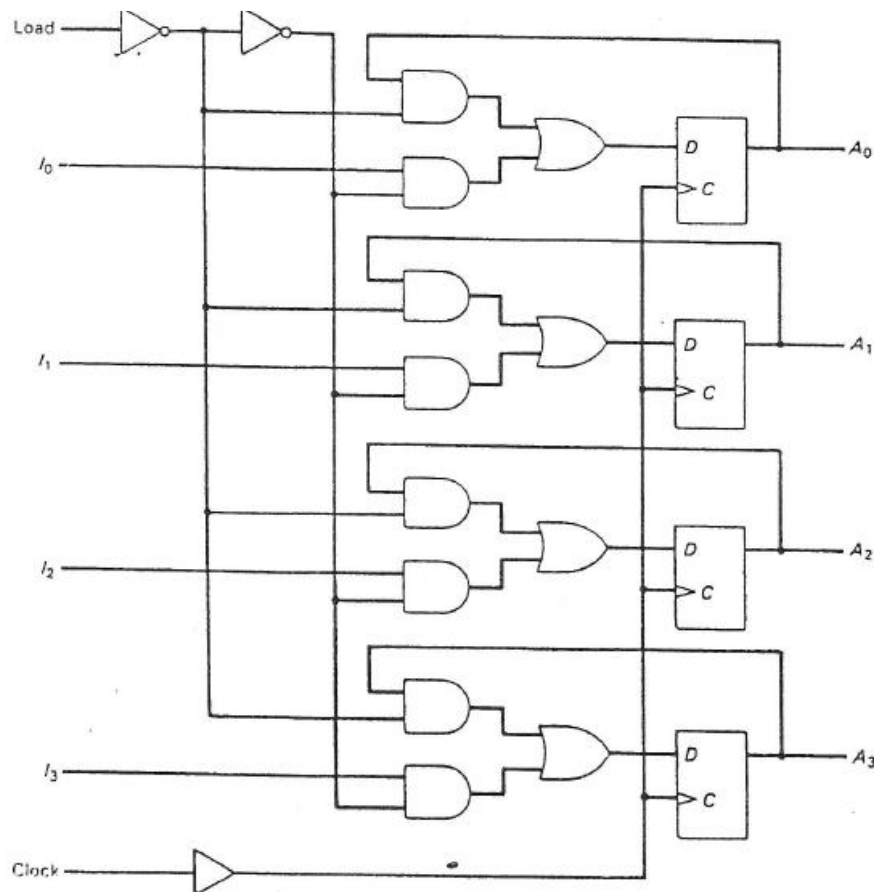


Fig.39 4-bit register with parallel load.

Normally, we do not want the transfer to occur with every clock pulse, but only under a predetermined condition. A conditional statement is symbolized with an if-then statement: IF ($T_1 = 1$) then ($R_2 \leftarrow R_1$).

It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. The control function is included in the statement as follows:

$$T_1 : R_2 \leftarrow R_1$$

It symbolizes the requirement that the transfer operation be executed by the hardware only if $T_1 = 1$.

Fig.40 shows the block diagram that depicts the transfer from R_1 to R_2 .

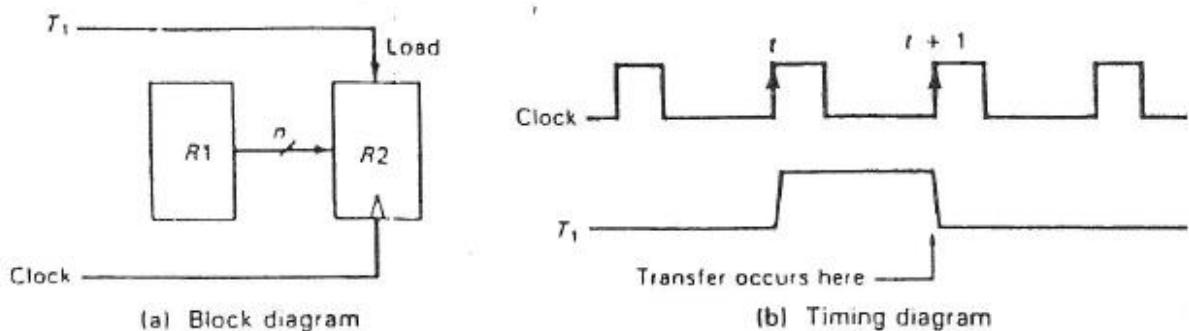


Fig.40 Transfer from R_1 TO R_2 when $T_1 = 1$.

5.1.1 Basic symbols for register transfer:

Symbol	Description	Example
Letters and numerals	Denotes a register	AR, R_2
parentheses	Denotes a part of a register	$R_2(0-7)$, $R_2(L)$
Arrow	Denotes transfer of information	$R_2 \leftarrow R_1$
Comma	Separates two microoperations	$R_2 \leftarrow R_1, R_1 \leftarrow R_2$
Square brackets []	specify an address for memory	$DR \leftarrow M[AR]$

The statement $T_3: R_2 \leftarrow R_1, R_1 \leftarrow R_2$, denotes an operation that exchanges the contents of two registers during one common clock pulse provided $T_3 = 1$. This Simultaneous operation is possible with Registers that have edge-Triggered flip-flops.

5.1.2 Multiplexer Selection

There are occasions when a register receives information from two different sources at different times.

$$\begin{aligned} T_1: R_0 &\leftarrow R_1 \\ T_1 T_2: R_0 &\leftarrow R_2 \end{aligned}$$

This specifies a hardware connection from two registers, R_1 and R_2 , to one common register R_0 . This type of operation requires a multiplexer to select between the two source registers according to the values of the timing variables as shown in Fig.41.

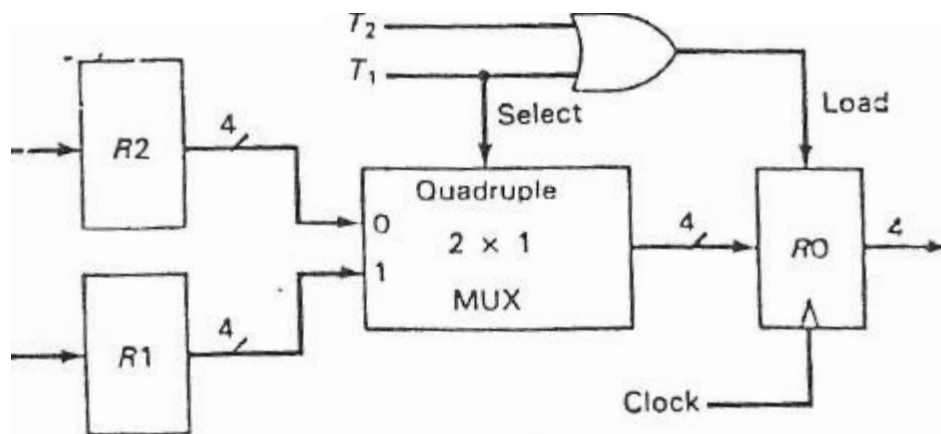


Fig.41 Block diagram for two register connections.

5.2 Bus Transfer

A more efficient scheme for transferring information between registers in a multiple register configuration is a bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals select which register will be the source and which will be the destination during each register transfer.

One way of constructing a common bus system is with multiplexer and a decoder.

The multiplexers select one source register whose binary information is then placed on the bus. The decoder selects one destination register to accept the information from the bus. The construction of a bus system for four registers is shown in Fig.42.

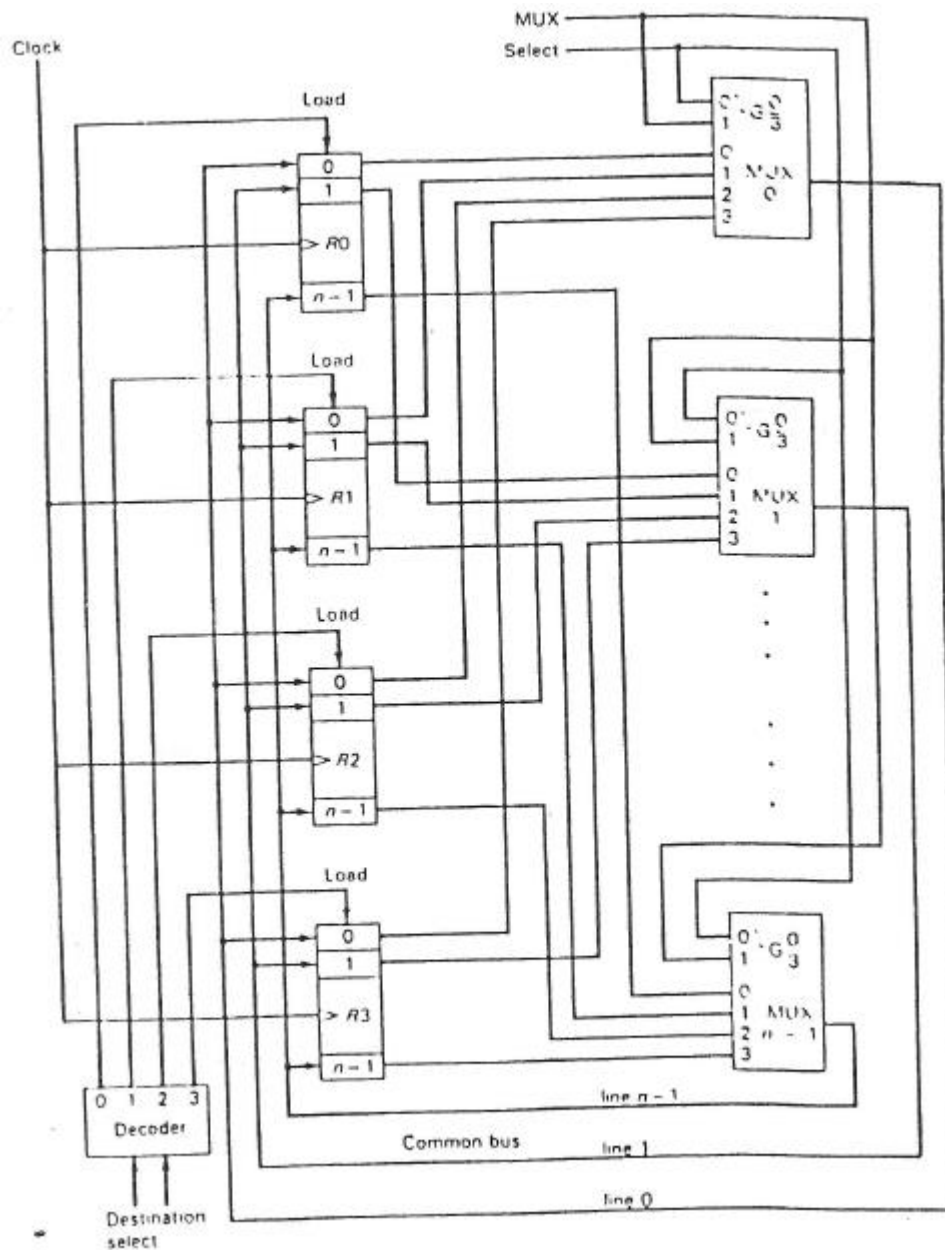


Fig.42 Bus system for four registers.

Each register has n -bits. The bits in the same significant position in each register are applied to a 4-to-1 line multiplexer to form one line of the bus. The n -lines formed by the common bus system are routed to the n -inputs of each register. The transfer of information from the bus into one destination register is accomplished by activating the load control input of the selected register.

As an example, consider the transfer given by $R_2 \leftarrow R_0$. The control variables that enables this transfer must select register R_0 as the source for the bus and register R_2 as the destination. The multiplexer MUX selects inputs must be binary 00. This causes bit 0 of R_0 to be applied to line 0 of the bus through MUX_0 , bit 1 of R_0 to line 1 of the bus through MUX_1 . This repeats for all other bus lines up to line $n-1$. Accordingly, the n -bit value of R_0 is placed on the common bus.

The destination select input must be binary 10. This activates output 2 of the decoder, which in turn activates the load input of R_2 . With the next clock transition, the contents of R_0 , being on the bus, are loaded into R_2 to complete the transfer.

5.2.1 Three-state Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A graphic symbol for a three-state buffer gate is shown in Fig.43.

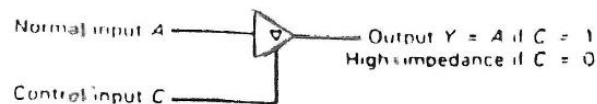


Fig.43 Graphic symbol for a Three-state buffer.

The construction of a bus system with three-state buffers is shown in Fig.44. The outputs of four buffers are connected together to form a single

bus line. The control inputs to the buffers determine which one of the four normal inputs will communicate with the bus line. One way to ensure that no more than one control input is active at any given time is to use a decoder as in the figure. When the enable input of the decoder is 0, all of its outputs are 0, and the bus line is in a high impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active depending on the binary value of the select inputs of the decoder.

There are occasions where it is necessary to employ a bidirectional bus system that can transfer information in both directions. A bidirectional bus can be constructed with three-state buffers to control the direction of information flow in the bus. One line of a bidirectional bus is shown in Fig.45. The bus control has two selection lines S_{in} for input transfer and S_{out} for output transfer.

These selection lines control two three-state buffer connected back to back. When $S_{in} = 1$ and $S_{out} = 0$, the bottom buffer is enabled and the top buffer is disabled.

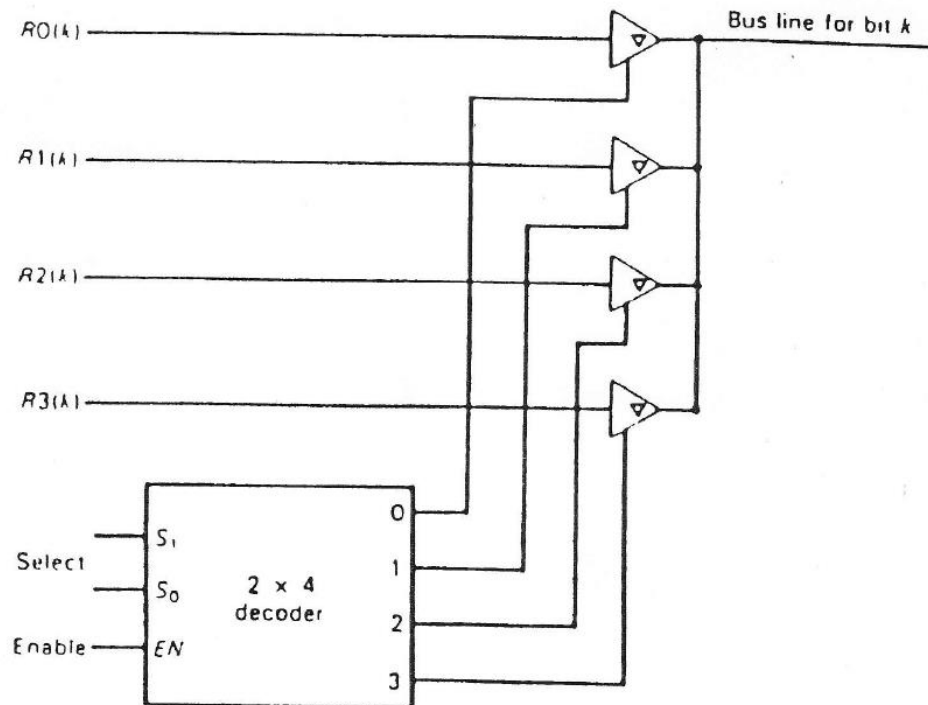


Fig.44 Bus line with three-state buffers.

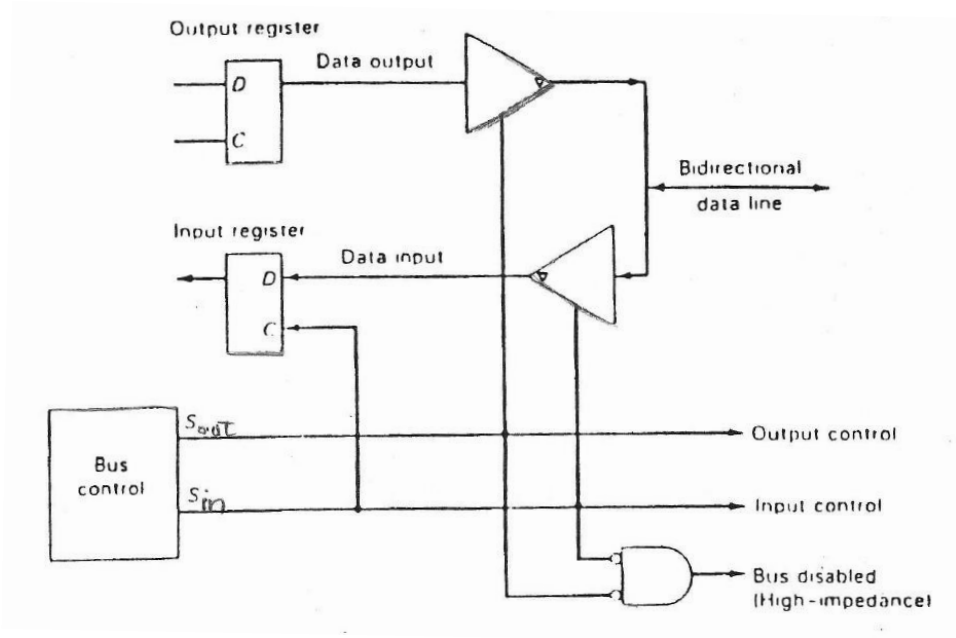


Fig.45 Bidirectional bus line with three-state buffers.

This forms a path for input data coming from the bus to pass through the bottom buffer and into the input of a flip-flop register.

For $S_{in} = 0$ and $S_{out} = 1$, the top buffer is enabled and the bottom buffer is disabled. This forms a path for the output data coming from a register in the system to pass through the upper buffer and out to the bus line. Making both control signals 0, the bus line is disabled. This condition will exist when an external source is using the common bus to communicate with some other external destination.

5.3 Memory Transfer

Consider a memory unit that receives the address from a register called address register AR. The data is transferred to another register called data register DR.

The Read operation can be stated as:

Read : $DR \leftarrow M [AR]$

This causes a transfer of information into DR from the selected memory word specified by the address in AR.

The Write operation is a transfer from DR to the selected memory word M.

Write : $M [AR] \leftarrow DR$

This causes a transfer of information from DR into the memory word selected by the address in AR.

In some systems, the memory unit receives address and data from many registers connected to common buses. Consider the case depicted in Fig.46.

The address for the memory comes from an address bus. Four registers are connected to this bus and any one may supply an address. The memory data is connected to a bidirectional data bus. The direction of information flow in the data bus is determined from the bus control inputs. For a read operation, the path is from memory to a data register. For a write operation, the path is from a data register to a memory.

The statement $M [A_1] \leftarrow D_2$ specifies a transfer of information from D_2 to a memory word selected by the address in A_1 (write operation).

The read operation in a memory with buses can be specified by the statement $D_0 \leftarrow M [A_3]$.

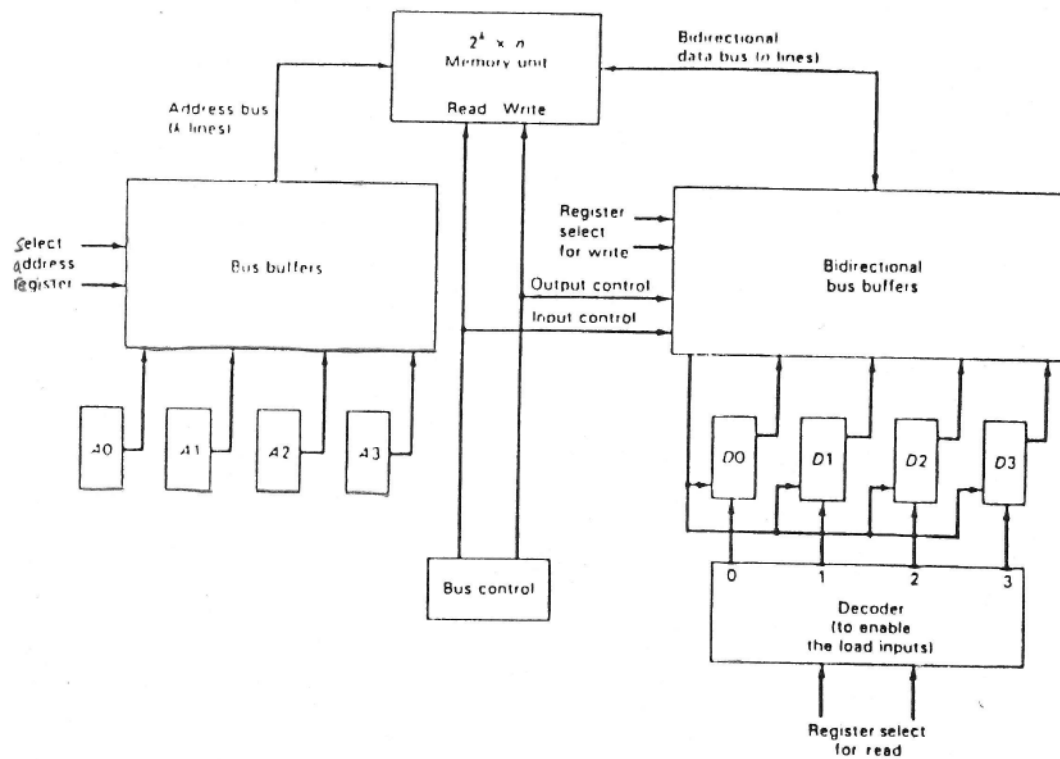


Fig.46 Memory unit connected to address and data buses.

Problems

[1] Show the block diagram of the hardware that implements the following register transfer statement:

$$T_3 : R_2 \leftarrow R_1, R_1 \leftarrow R_2$$

[2] The outputs of four registers R_0 , R_1 , R_2 , and R_3 are connected through multiplexers to the inputs of a fifth register R_5 . Each register is eight bit long. The required transfers are dictated by four timing variables T_0 through T_3 as follows:

$$T_0 : R_5 \leftarrow R_0$$

$$T_1 : R_5 \leftarrow R_1$$

$$T_2 : R_5 \leftarrow R_2$$

$$T_3 : R_5 \leftarrow R_3$$

The timing variables are mutually exclusive and only one variable can be equal to 1 at any given time while the other three are equal to 0. Draw a block diagram showing the hardware implementation of the register transfers. Include the connections necessary from the four timing variables to the selections lines of the multiplexers and to the load input of register R_5 .

[3] Show that the statement $R_1 \leftarrow R_1 + R_1$ is the same as the logical shift left of register R_1 .

[4] Let S_1 and S_0 be the selection variables for the multiplexer in the bus system, and D_1 and D_0 are the selection variables for the destination decoder.

a. Determine the transfer that occurs when the control variables S_1 S_0 D_1 D_0 equal:

1. 0001

2. 0110

3. 1111

b. Give the 4-bit selection for the following transfers:

1. $R_0 \leftarrow R_1$

2. $R_2 \leftarrow R_1$

3. $R_3 \leftarrow R_2$

[5] Draw a diagram of a bus system similar to that in (4), but use three-state buffers and decoder instead of the multiplexers.

[6] The following memory transfer are specified for the system in the (lectures).

a. $D_2 \leftarrow M[A_3]$

b. $M[A_1] \leftarrow D_0$

Specify the memory operation and determine the binary selection variables or the two bus buffers and the decoder.

CHAPTER SIX

INPUT – OUTPUT AND COMMUNICATIONS

No computer, regardless of its size, speed, computing capabilities, or other sophisticated features, is very useful unless it can communicate with outside world (i.e. with other equipment in the system that are called peripherals). This communication involves raw data coming into the computer from the peripherals and the transfer of processed data (or control signals) from the computer to the peripherals.

The computer and the peripherals seldom operate at the same speed. The peripherals, which could be electromechanical devices, usually operate at much slower speeds than the all-electronic CPU. These speeds and/or timing differences somehow must be reconciled. Furthermore, the data formats of the peripherals may be different from the format used by the computer. Some means of format conversion is needed.

The input/output system includes IO devices (peripherals), control unit for these devices, and the software designed to carry out IO operations.

6.1 IO Techniques

IO operations may be completely controlled by the CPU, i.e., the CPU executes programs that initiate, direct, and terminate the IO operations. Accordingly, the computer is said to be using programmed IO.

With a fairly modest increase in hardware complexity, the IO device can be provided with the ability to transfer a block of information to or from main memory without CPU intervention. This requires that the IO device (or its controller) be capable of generating memory address and transferring data to or from the system bus, i.e., it must be a bus master. The CPU is still responsible for initiating each block transfer. This type of input output capability is called direct memory access (DMA).

The IO device or its controller can also be provided with circuits enabling it to request service from the CPU, i.e., execution of a specified program to service the IO device. This type of request is called an interrupt. An interrupt capability frees the CPU from the task of periodically testing IO device status.

Most computers now have DMA and interrupt facilities. A DMA has partial control of IO operations. Essentially complete control of IO

operations can be relinquished by the CPU if an IO processor (IOP) or channel is introduced. Like a DMA controller, an IOP has direct access to main memory and can interrupt the CPU, however, it can also execute programs directly. These programs, called IO programs, may employ an instruction set different from that of the CPU-one which is oriented toward IO operations. Usually an IOP is connected to the devices it controls by a separate bus system, called the IO bus or IO interface.

Naturally, the various processor level components (CPUs, IOPs, main memory, IO or peripheral devices) of a computer system are interconnected by buses.

Many bus organizations are possible. Two common types are depicted in Fig.47.a,b. In (a) only two units can communicate via the system bus. Large computer systems with separate IO processors frequently employ the dual bus system in (b).

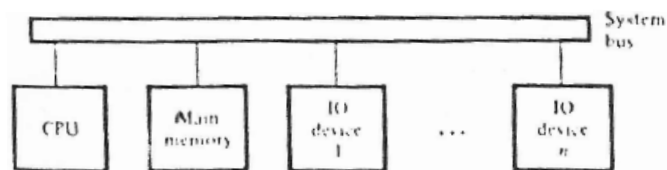


Fig.47.a. Single bus, system bus is shared by all components.

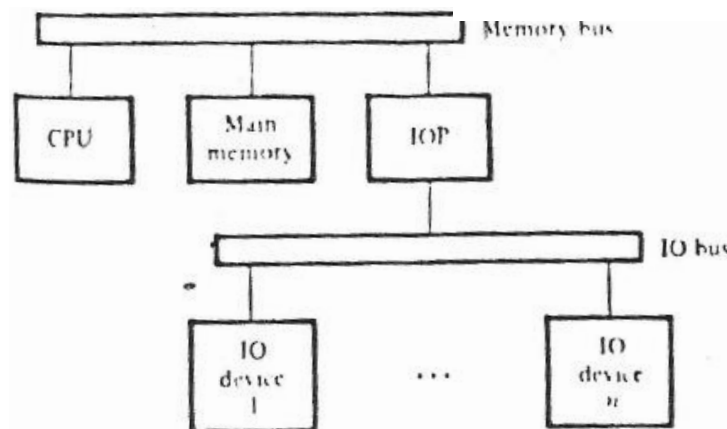


Fig.47.b. Separate memory and IO buses.

6.1.1 Programmed IO

This is a method for controlling IO operations, which is included in most computers. It is particularly useful in small low-speed systems where hardware costs must be minimized. It requires that all IO operations be executed under the direct control of the CPU, i.e., every data-transfer operation involving an IO device requires the execution of an instruction by the CPU. Typically, the transfer is between a CPU register, e.g., the accumulator, and a buffer register connected to the IO device. The input output device does not have direct access to main memory.

a. IO addressing:

In systems with programmed IO, IO devices, main memory, and the CPU normally communicate via a common shared bus (the system bus). Each junction between the system bus and IO device is called an IO port and is assigned a unique address. The IO port includes data buffer register. Some machines (68000) use memory-mapped IO, in which part of the main memory address space is assigned to IO ports. Other organization, as shown in Fig.48 is sometimes called IO-mapped IO, the memory and IO address space are kept separate. One of the programmed IO is shown in Fig.49.

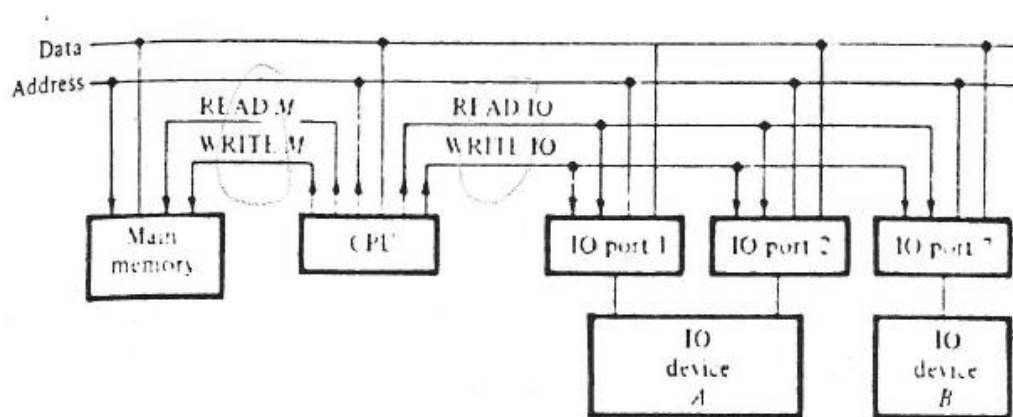


Fig.48 IO-mapped IO.

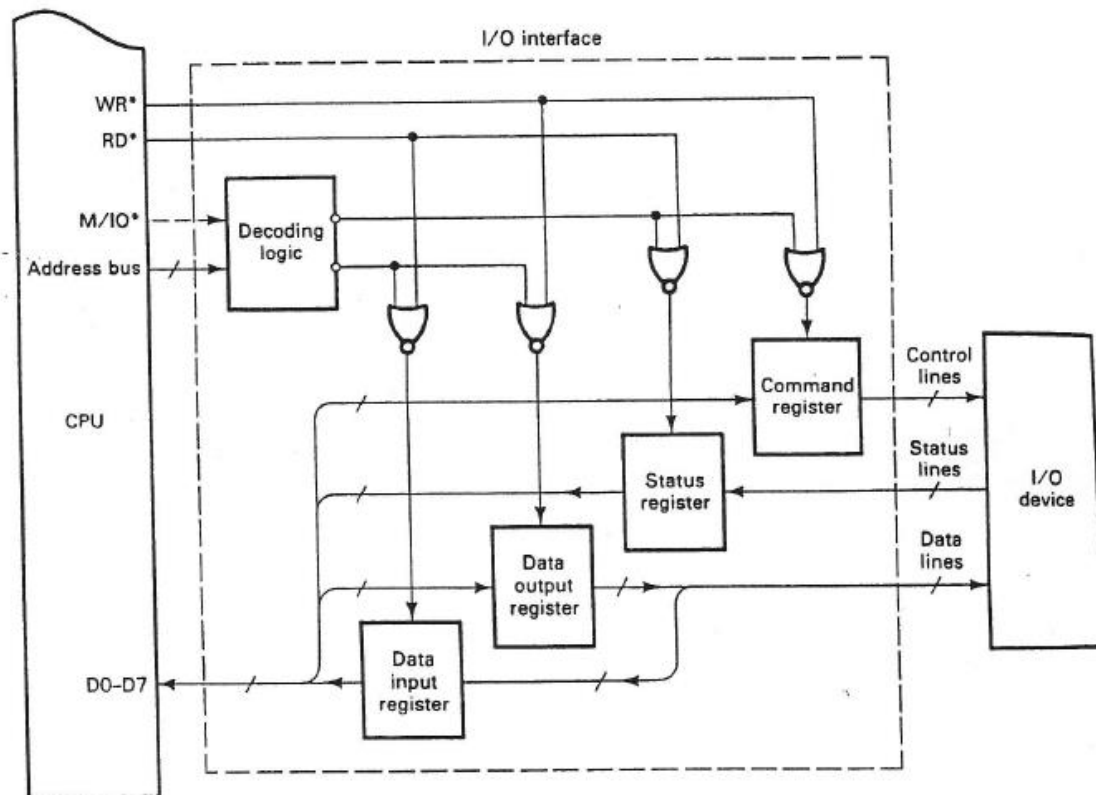


Fig.49 Simplified diagram of an I/O interface.

Notice from Fig.49, that all the registers are connected to the CPU data bus. The command and data output registers are write only registers. The status and data input registers are read-only registers. Hence, a total of two addresses is all we need. Whether implemented with real registers or actually representing buffers, the four I/O registers are also known as I/O ports.

In fact, this scheme represents a particular form of programmed I/O known as polled I/O. The name stems from the fact that prior to performing a data transfer, the CPU "polls" the IO device to check its readiness. Status checking does not make sense in the case of very simple IO devices (like switches or visual indicators). This particular form of programmed IO is called direct I/O.

An example of direct IO is shown in Fig.50. As an example of the machines that use separate I/O address space is the 8088 & 8086 microprocessors, which have a separate IO address space of 64KB.

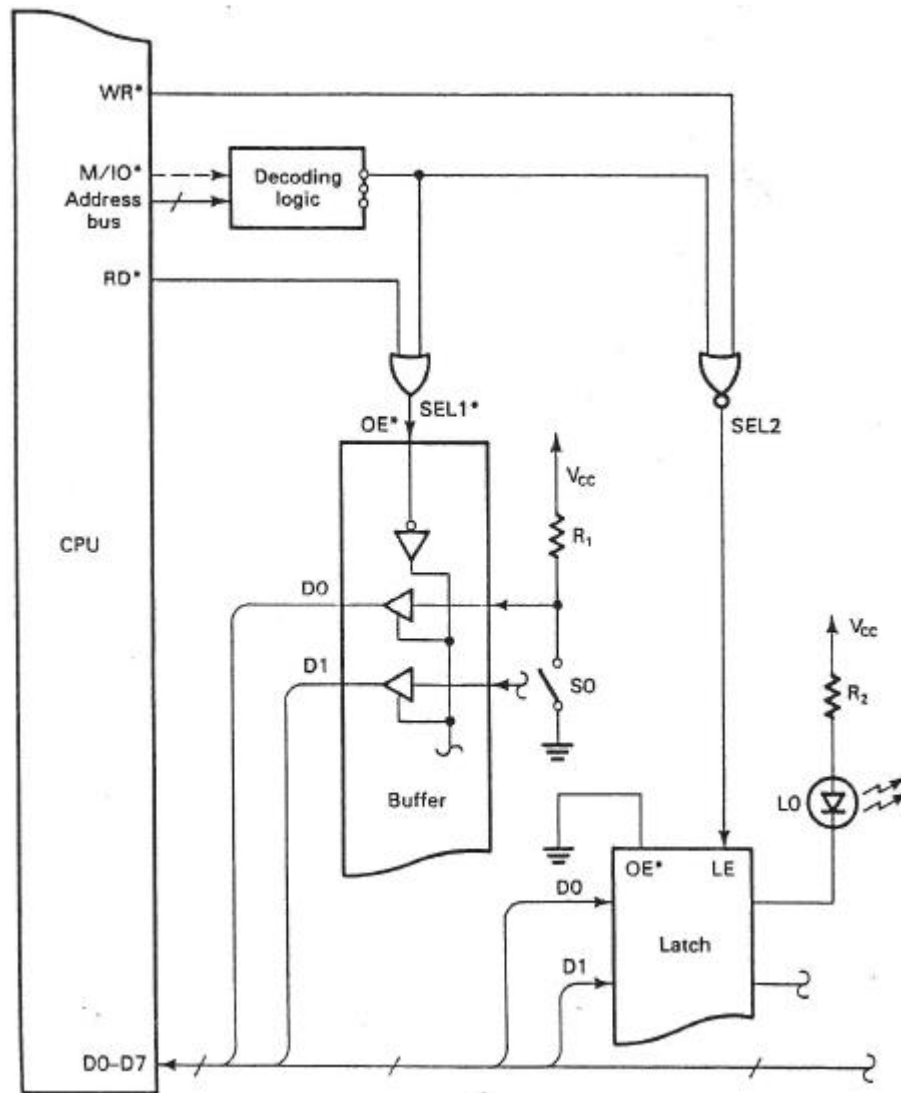


Fig.50 Example of direct IO.

b. IO instructions

Programmed IO can be implemented by as few as two IO instructions. For example, the Intel 8085 has two main IO instructions (INX & OUTX). In programmed IO systems, the CPU is usually programmed to test the IO device status before initiating a data transfer. Often the status can be specified by a single bit of information that the IO device can make available on a continuous basis, e.g., by setting a flip-flop connected to the data lines at some IO ports.

If programmed IO is the primary method of controlling IO devices in a computer, additional IO instructions may be provided to augment the simple IN and OUT instructions. For ex., the DEC PDP-8 has modified the CPU program counter based on the test outcome. TSK, can be implemented as shown in Fig.51. On decoding TSK, a signal called TEST STATUS is sent by the CPU to the IO device. If the device status flag is set, a return pulse is sent on the SKIP line which is used to increment the program counter, thereby skipping the next instruction.

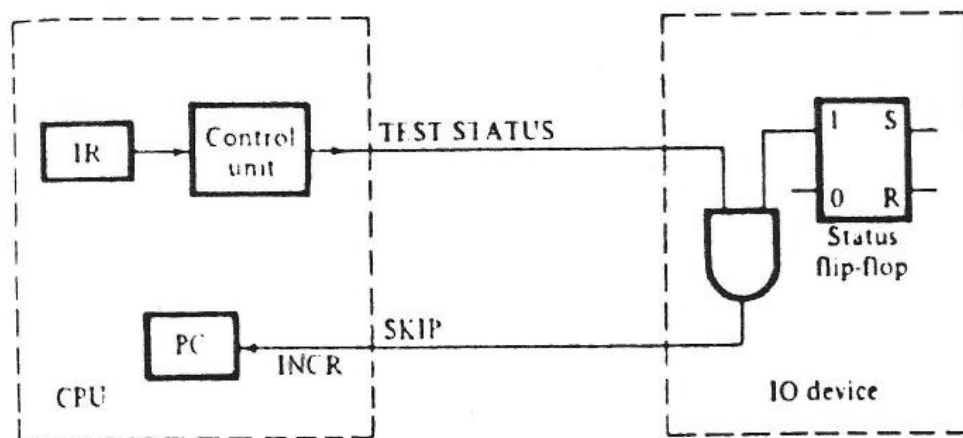


Fig.51 Implementation of a test status and skip (TSK) IO instruction

c. IO interface circuits

The connection of IO devices to a computer system is greatly facilitated by the use of standard circuit packages variously known as IO interface circuits, peripheral interface adapters, etc.

The simplest interface circuit is one-word buffer register that acts as an IO port. It is assigned a unique address and is accessed in essentially the same way as a main-memory location. This circuit is particularly useful for parallel (word-by-word) IO communication. Another useful class of interface circuits are universal asynchronous receiver transmitter UARTs, which allow easy connection to the computer of IO devices that employ serial (bit-by-bit) communication, for example, a modem as shown in Fig.52.

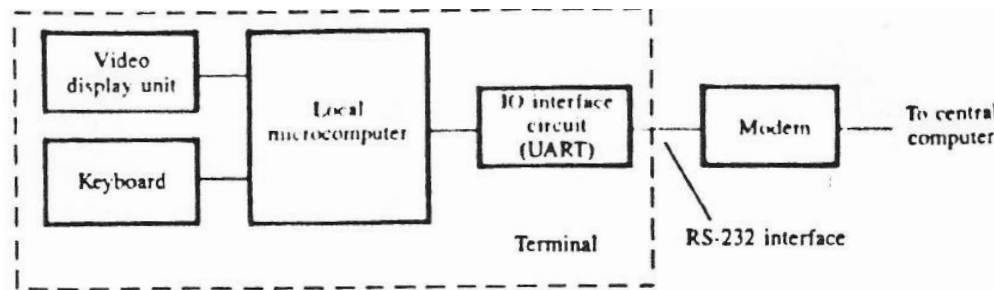


Fig.52 Architecture of a typical terminal.

A UART is a programmable shift register that transforms serial data streams into parallel data streams, and vice versa. The advent of microprocessors has greatly stimulated the design of powerful general-purpose interface circuits, (ex. 8255-PPI, Peripheral Interface Adapter PIA as 6820 IO interface).

6.1.2 DMA and Interrupts

The programmed IO method has two main drawbacks:

1. IO transfer rates are limited by the speed with which the CPU can test and service an IO device.
2. The time that the CPU spends testing IO device status and executing IO data transfers can often be better spent on other processing tasks.

DMA and Interrupt circuits are used to increase the speed of IO operations and eliminate most of the role played by the CPU in such operations. In each case, special control lines to which we assign the generic names DMA REQUEST and INTERRUPT REQUEST go from the IO devices to the CPU. Signals on these lines cause the CPU to suspend its current activities at an appropriate breakpoint and attend to the DMA or interrupt request. DMA further allows IO data transfer to take place without the execution of IO instructions by the CPU.

a. Direct Memory Access (DMA)

The essential elements of a DMA system are shown in Fig.53. The IO device is connected to the system bus via a special interface circuit, a DMA controller, which contains a data buffer register IODR as in the programmed IO case, but in addition there is an address register IOAR and a data count register DC. These registers allow the DMA controllers to transfer data to or from a contiguous region of the main memory. IOAR is used to store the address of the next word to be transferred. It is automatically incremented after each word transfer. The data count register DC stores the number of words that remain to be transferred. It is automatically decremented after each transfer and tested for zero. When the DC reaches zero, the DMA transfer halts.

The controller is normally provided with an interrupt capability, in which case it sends an interrupt to the CPU to signal the end of the data transfer.

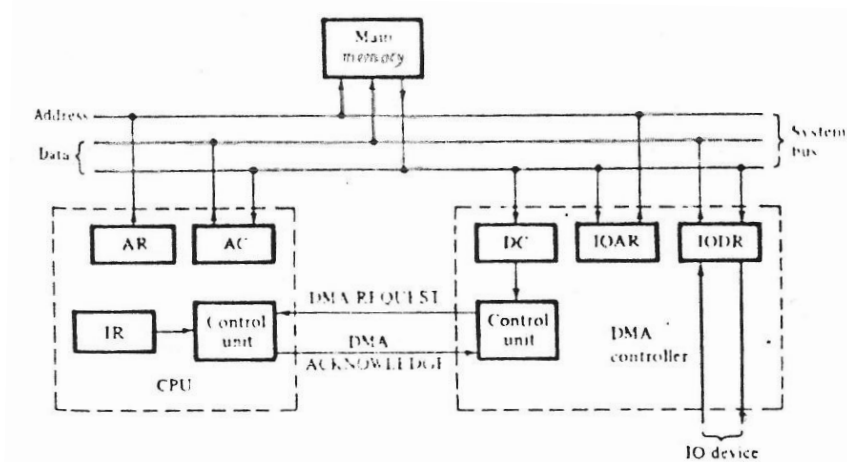


Fig.53 Circuitry required for DMA.

The CPU may be placed in an idle state in a variety of ways; one common extensively used in microprocessors is to disable the buses through special control signals. Fig.54 shows two control signals in a CPU that facilitate the DMA transfer. The bus request (BR) input is used by the DMA controller to request from the CPU to relinquish control of the buses. As long as the BG line is active, the CPU is idle and the buses are disabled.

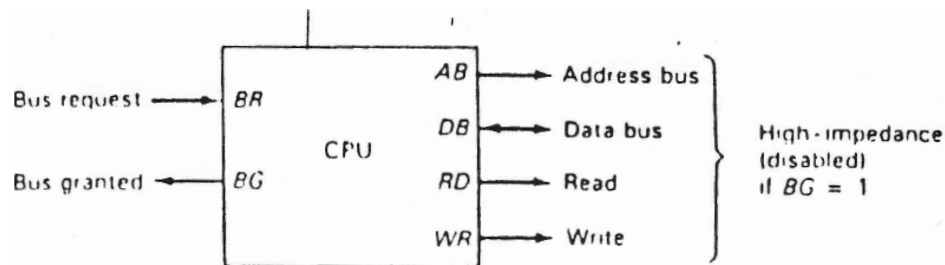


Fig.54 CPU bus control signals.

DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Fig.55. The DMA has its own address which activates the DMA select (DS) and the register select (RS) lines.

The CPU initializes the DMA through the data bus. Once the DMA receives the start control bit, it can start the transfer between the peripheral device and the memory.

When a peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled.

The DMA then puts the current value of its address register onto the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus, the DMA controls the read or write operations and supplies the address for the memory.

The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

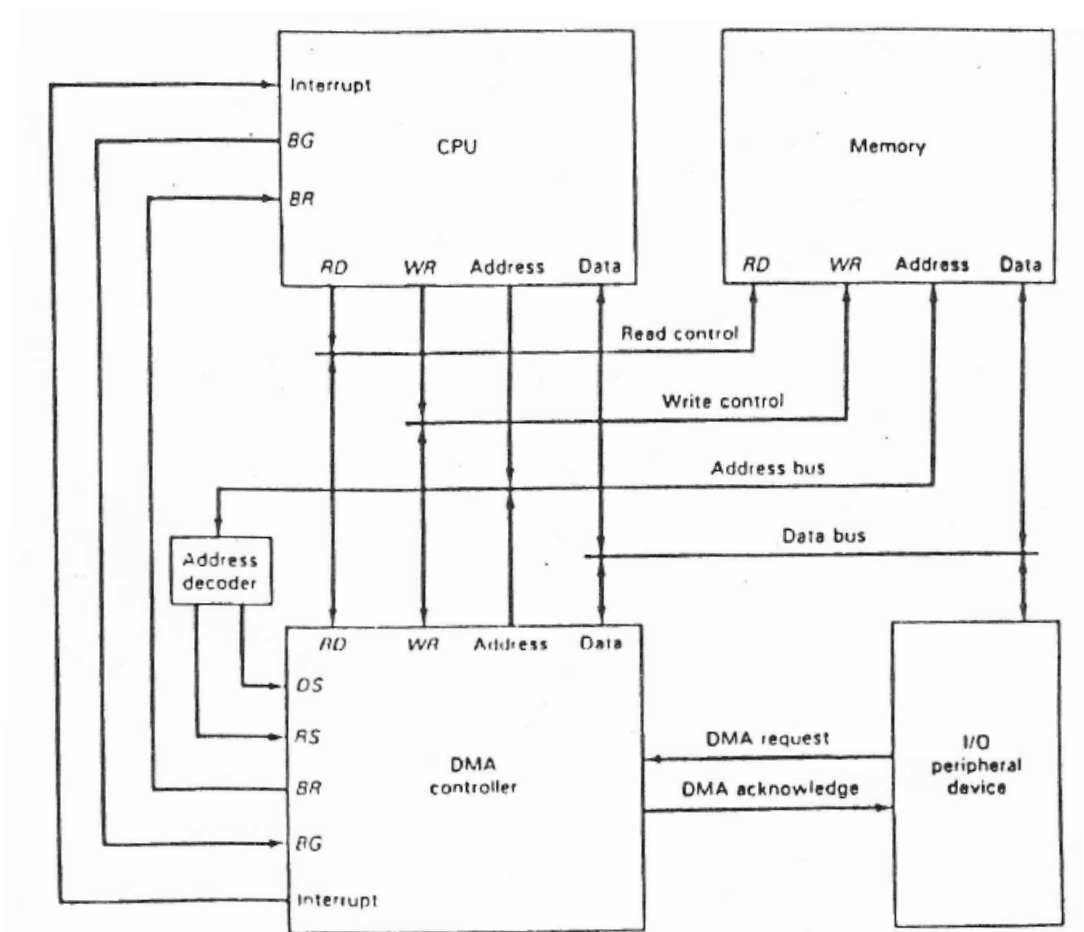


Fig.55 DMA Transfer in a computer system.

b. Interrupt

The term interrupt is used in a loose sense for any infrequent or exceptional event that causes a CPU to make a temporary transfer of control from its current program to another program that services the event. Interrupts greatly increase the performance of the computer by allowing the IO devices direct and rapid access to the CPU and by freeing the CPU from the task of continually testing the status of its IO devices.

Interrupts are used primarily to request the CPU to initiate a new IO operation, to signal the completion of an IO operation, and to signal the occurrence of hardware or software errors.

Interrupts generated internally by a CPU are called traps, and result from such programming errors as an attempt to divide by zero.

The basic method of interrupting the CPU is by activating a control line (interrupt request) that connects the interrupt source to the CPU. The interrupt signal is then stored in a CPU, usually at the end of every instruction cycle.

On recognizing the presence of the interrupt, the CPU must execute a specific interrupt servicing program. Normally, each interrupt source will require execution of a different program; the CPU must therefore determine or be given the address in main memory of the specific address program to be used. A further problem is caused by the presence of two or more interrupt requests at the same time. Priorities must be assigned to the interrupts.

Priority Interrupt

Establishing the priority of simultaneous interrupts can be done by software or hardware. The disadvantage of software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the IO device. In this situation, a hardware priority interrupt unit can be used to speed up the operation.

The hardware priority function can be established either by a serial or parallel connection of interrupt lines. The serial connection is also known as the daisy chain methods.

Daisy chain priority

The Daisy chain method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with highest priority is placed in the 1st position, followed by lower priority devices. This method of connection between three devices, and the CPU is shown in Fig.56.

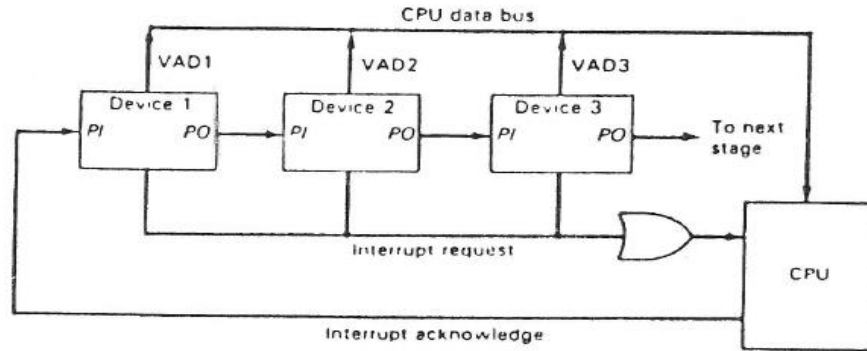


Fig.56 Daisy chain priority Interrupt.

The interrupt request line is common to all devices and forms a wired logic connection. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input.

The acknowledge signal passes on the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle. Fig.57 shows the internal logic that must be included within each device when connected in the daisy chain scheme.

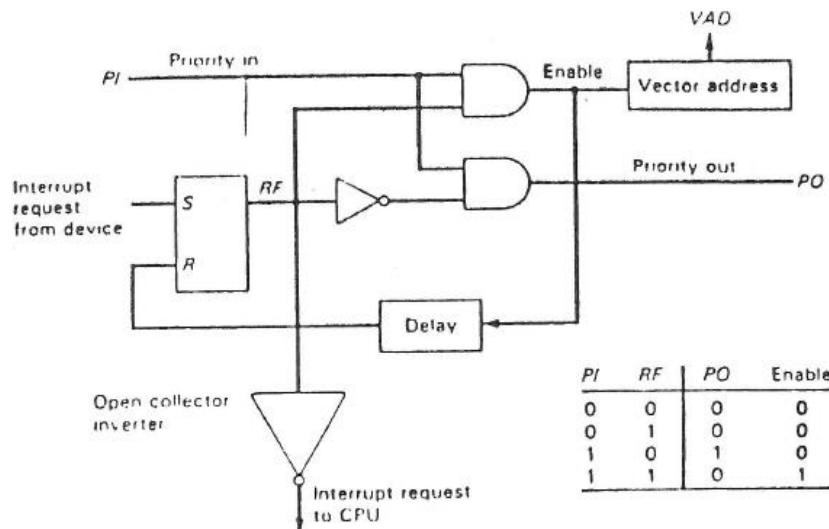


Fig.57 Stage of the daisy chain priority arrangement.

Parallel Priority Hardware

The parallel priority interrupt method uses a register with bits set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register to control the status of each interrupt request.

The mask register can be programmed to disable lower priority interrupts while a higher priority device is being serviced. It can also provide a facility that allows a high priority device to interrupt the CPU while a lower priority device is being serviced. The priority logic for a system of four interrupt sources is shown in Fig.58.

The priority encoder is a circuit that implements the priority function as shown in the truth table and the circuit of Fig.59. Input D_3 has the highest priority, and so regardless of the values of other inputs, when this input is 1 the output is $A_1A_0 = 11$, and so on. The interrupt output labelled V is equal to 1 when one or more inputs are equal to 1. If all inputs are 0, V is 0 and the other two outputs of the encoder are not used. This is because the vector address is not transferred to the CPU when $V = 0$. The output of the priority encoder is used to form part of the vector address for the interrupt source. The other bits of the vector address can be assigned any values. (Here 0's are assigned for the six bits, accordingly, the 8-bit equal to the source IO are 0, 1, 2, &3).

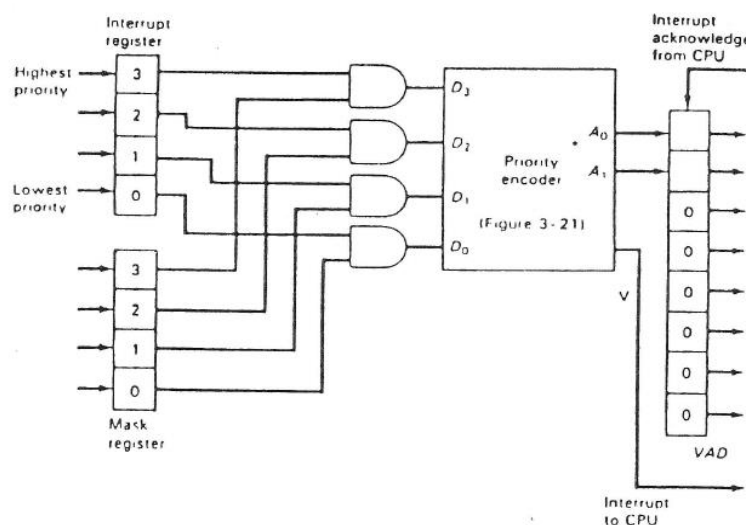


Fig.58 Priority Interrupt Hardware.

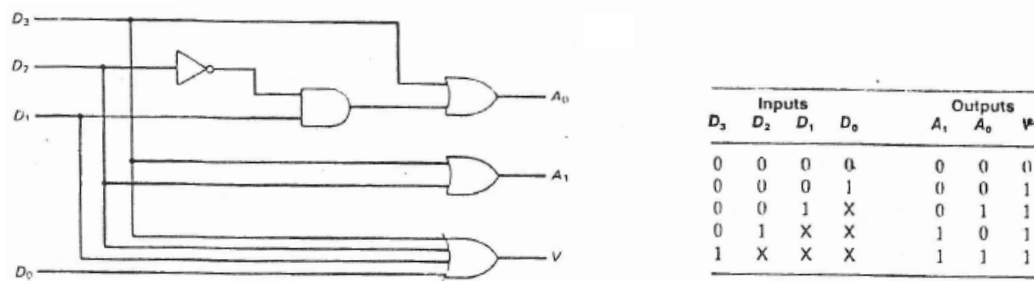


*Fig.59.a. Logic diagram of a 4-bit priority encoder.
b. Truth table for priority encoder.*

6.2 Input-Output Processor (IOP)

A DMA has partial control of IO operations. Complete control of IO operations can be relinquished by the CPU if an IOP or channel is introduced. Like a DMA controller, an IOP has direct access to main memory and interrupt the CPU.

The IOP is similar to a CPU except that it is designed to handle the details of IO processing. Unlike the DMA controller that must be set up entirely by the CPU the IOP can fetch and execute its own instructions.

IOP instructions are specially designed to facilitate IO transfers. In addition, the IOP can perform other processing tasks such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in Fig.60. The data formats of PDs differ from memory and CPU data formats. The IOP must structure data words from many different sources. For ex., it may be necessary to take four bytes from an Input device and packs them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. Communication with the memory is similar to the DMA methods. In most computer systems, the CPU

is the master, while the IOP is a slave processor. Instruction that are read from memory by an IOP are sometimes called commands, to distinguish them from instruction that are read by the CPU.

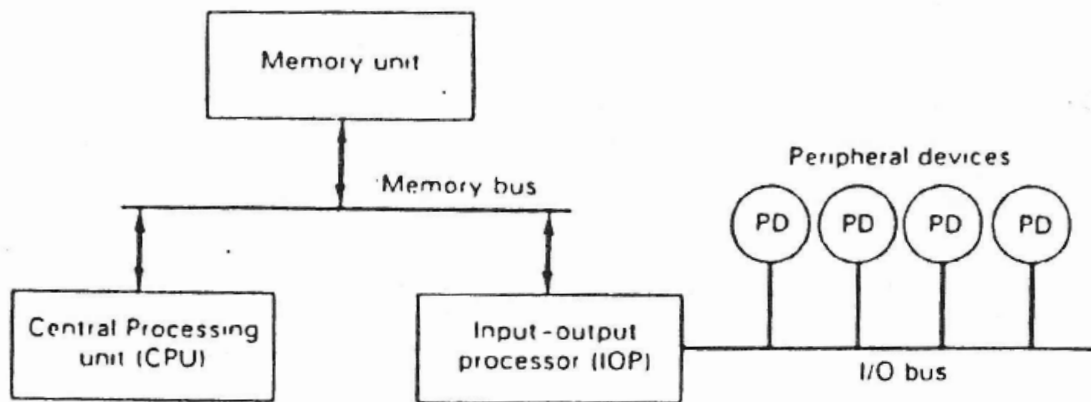


Fig.60 Block diagram of a computer with IO processor.

6.3 Handshaking

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One method being used is to accompany each data item responds with another control signal to acknowledge receipt of the data. This kind of arrangement between two independent units is referred to as handshaking.

The basic principle of the two-wire handshaking procedure of data transfer is described in Fig.61.a, b, and c.

The initial state is when both the data ready and acknowledge are disabled and in the 00 state. The subsequent states are 10, 11, and 01. The handshaking scheme provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units.

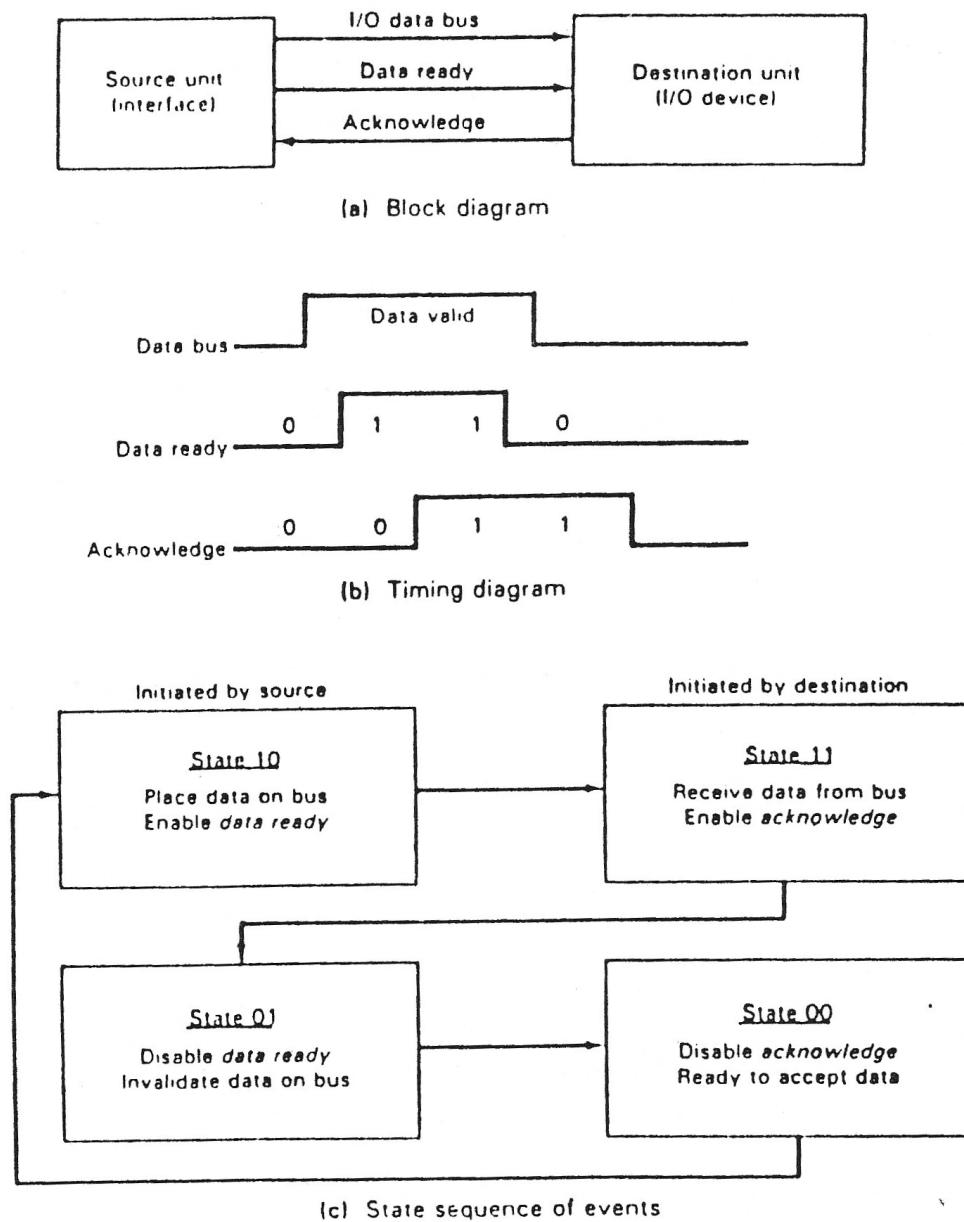


Fig.61. Asynchronous transfer using handshaking.

- a. Block diagram.
- b. Timing diagram.
- c. State sequence of events.

Problems

[1] A computer consists of a CPU and an IO device D connected to main memory M via a one-word shared bus. The CPU can execute a maximum of 10 instructions/sec. An average instruction requires five machine cycles, three of which use the memory bus. A memory read or write operation uses one machine cycle. Suppose that the CPU is continuously executing "background" programs that require programs that require 95% is to be used to transfer very large blocks of data to and from M.

- a. If programmed IO is used and each one-word IO transfer requires the CPU to execute two instructions, estimate the maximum IO data-transfer rate $T_{\max}$ possible through D.
- b. Estimate $T_{\max}$ if DMA transfer is used.

[2] Is it necessary to have a separate IO chip for each peripheral in the system? If not, how would this be handled? Where would any additional circuitry be located?

[3] A CPU has a four MB common address space for both memory and IO. It is to be used in an application where memory capacity will never exceed 2MB.

- a. How would you divide the address space between memory and IO so that, the distinction between memory and IO addresses (by the hardware) becomes very simple?
- b. Assume we need a separate select signal for each 256 KB of memory. Show how you would generate the memory select signals.

[4] What happens in the daisy chain priority interrupt, Fig.57, when device 1 requests an interrupt after device 2 has sent an interrupt request to the CPU but before the CPU responds with the interrupt acknowledge?

[5] It is necessary to transfer 256 words from a magnetic disk to a memory section starting from address 1230. The transfer is by means of a DMA.

- a. Give the initial values that the CPU must transfer to the DMA controller.

- b. Give the step-by-step account of the actions taken during the input of the 1st two words.

[6] Design a parallel priority interrupt hardware for a system with eight interrupt sources.

[7] The three outputs xyz, from the priority encoder are used to provide an 8-bit vector address in the form 101xyz00. List the eight addresses starting from the one with the highest priority.

CHAPTER SEVEN

PARALLEL PROCESSING

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel. This approach is taken further with superscalar organization, which exploits instruction-level parallelism.

With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel.

As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to enhance performance and, in some cases, to increase availability. After an overview, this chapter looks at some of the most prominent approaches to parallel organization. First, we examine symmetric multiprocessors (SMPs), one of the earliest and still the most common example of parallel organization. In an SMP organization, multiple processors share a common memory. This organization raises the issue of cache coherence, to which a separate section is devoted. Then we describe clusters, which consist of multiple independent computers organized in a cooperative fashion. Next, the chapter examines multithreaded processors and chip multiprocessors. Clusters have become increasingly common to support workloads that are beyond the capacity of a single SMP. Another approach to the use of multiple processors that we examine is that of nonuniform memory access (NUMA) machines. The NUMA approach is relatively new and not yet proven in the marketplace, but is often considered as an alternative to the SMP or cluster approach. Finally, this chapter looks at hardware organizational approaches to vector computation. These approaches optimize the ALU for processing vectors or arrays of floating-point numbers. They are common on the class of systems known as supercomputers.

7.1 Multiple Processor Organization

A taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer systems:

- **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.
- **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category.
- **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.
- **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fit into this category.

This taxonomy of parallel processor architectures is shown in Fig.62.

With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation. MIMDs can be further subdivided by the means in which the processors communicate. If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory. The most common form of such system is known as a symmetric multiprocessor (SMP). In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each

processor. A more recent development is the nonuniform memory access (NUMA) organization. As the name suggests, the memory access time to different regions of memory may differ for a NUMA processor.

A collection of independent uniprocessors or SMPs may be interconnected to form a cluster. Communication among the computers is either via fixed paths or via some network facility.

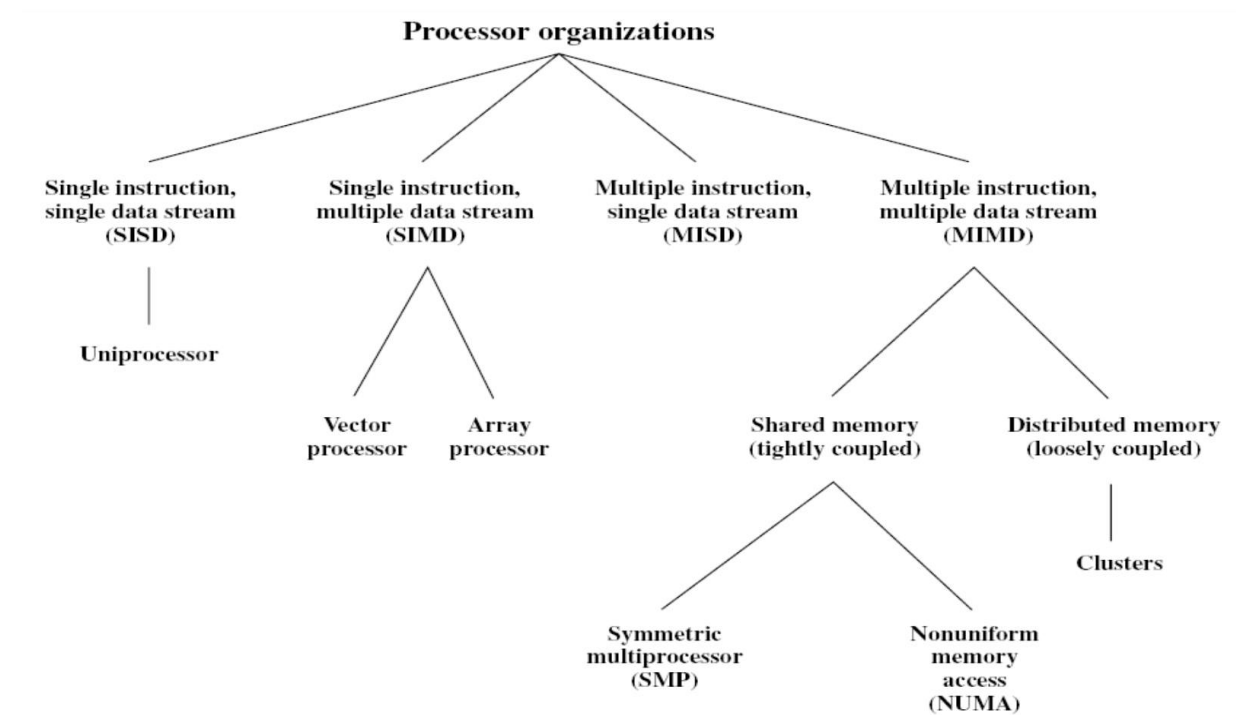


Fig.62. A taxonomy of parallel processor architectures

Parallel Organizations

Fig.63 illustrates the general organization of the taxonomy of Fig.62. Fig.63a shows the structure of an SISD. There is some sort of control unit (CU) that provides an instruction stream (IS) to a processing unit (PU). The processing unit operates on a single data stream (DS) from a memory unit (MU). With an SIMD, there is still a single control unit, now feeding a single instruction stream to multiple PUs. Each PU may have its own dedicated memory (illustrated in Fig.63b), or there may be a shared memory. Finally,

with the MIMD, there are multiple control units, each feeding a separate instruction stream to its own PU. The MIMD may be a shared-memory multiprocessor (Fig.63c) or a distributed-memory multicomputer (Fig.63d).

The design issues relating to SMPs, clusters, and NUMAs are complex, involving issues relating to physical organization, interconnection structures, interprocessor communication, operating system design, and application software techniques. Our concern here is primarily with organization, although we touch briefly on operating system design issues.

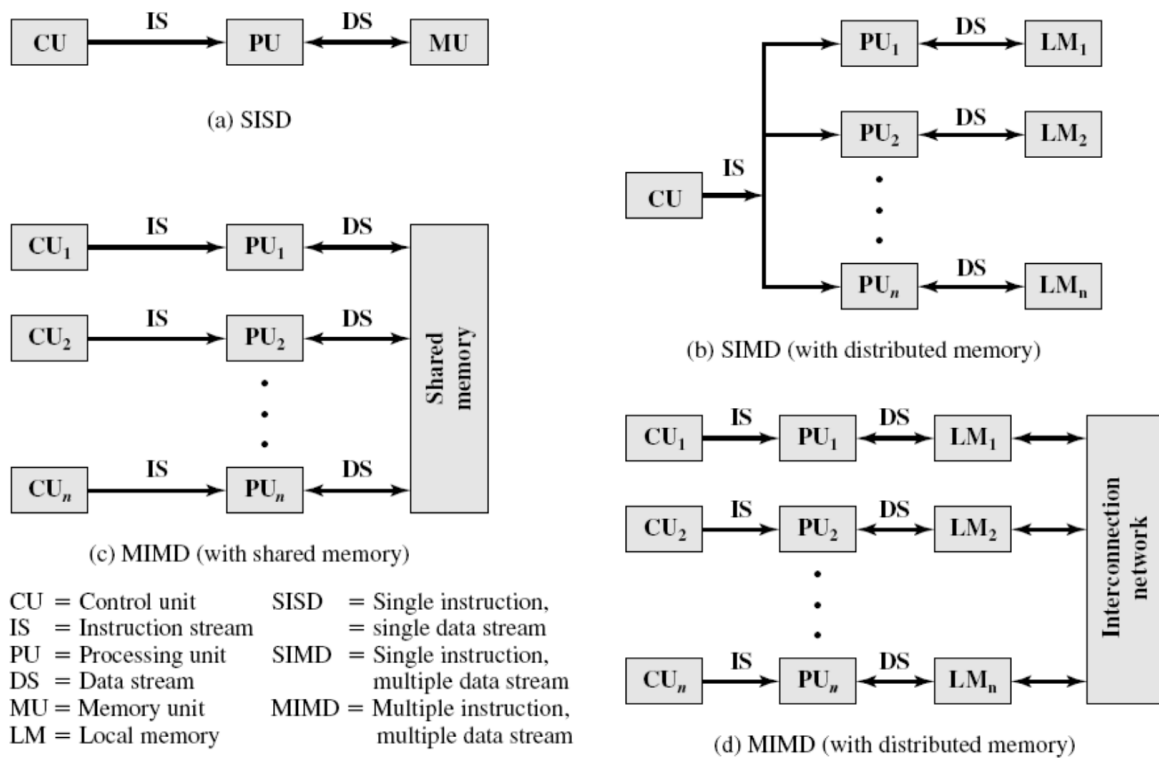


Fig.63. Alternative computer organizations

7.2 Symmetric Multiprocessors:

Until fairly recently, virtually all single-user personal computers and most workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced systems with an SMP

organization. The term *SMP* refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture. An SMP can be defined as a standalone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions (hence the term symmetric).
5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

Points 1 to 4 should be self-explanatory. Point 5 illustrates one of the contrasts with a loosely coupled multiprocessing system, such as a cluster. In the latter, the physical unit of interaction is usually a message or complete file. In an SMP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

The operating system of an SMP schedules processes or threads across all of the processors. An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.
- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does

not halt the machine. Instead, the system can continue to function at reduced performance.

- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronization among processors.

Organization

Fig.64 depicts in general terms the organization of a multiprocessor system. There are two or more processors. Each processor is self-contained, including a control unit, ALU, registers, and, typically, one or more levels of cache. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism. The processors can communicate with each other through memory (messages and status information left in common data areas). It may also be possible for processors to exchange signals directly.

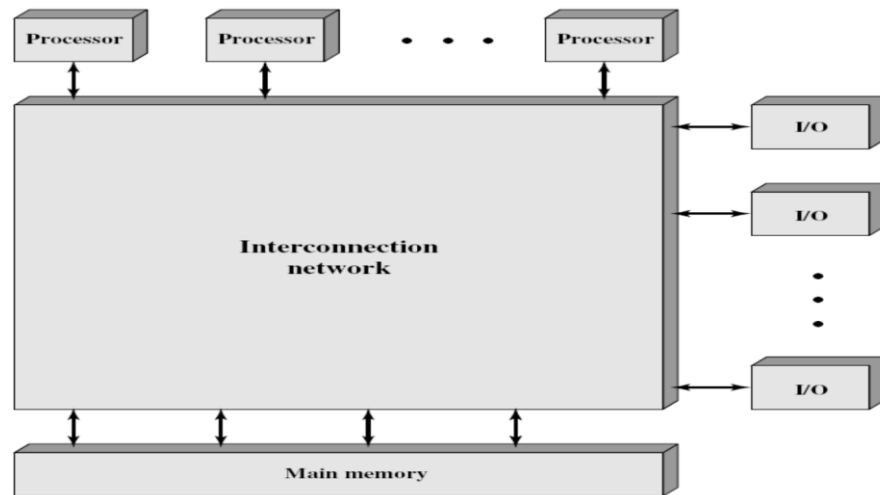


Fig.64. Generic block diagram of a tightly coupled multiprocessor

The most common organization for personal computers, workstations, and servers is the time-shared bus. The time-shared bus is the simplest mechanism for constructing a multiprocessor system (Fig.65). The structure and interfaces are the same as for a single-processor system that uses a bus interconnection. The bus consists of control, address, and data lines.

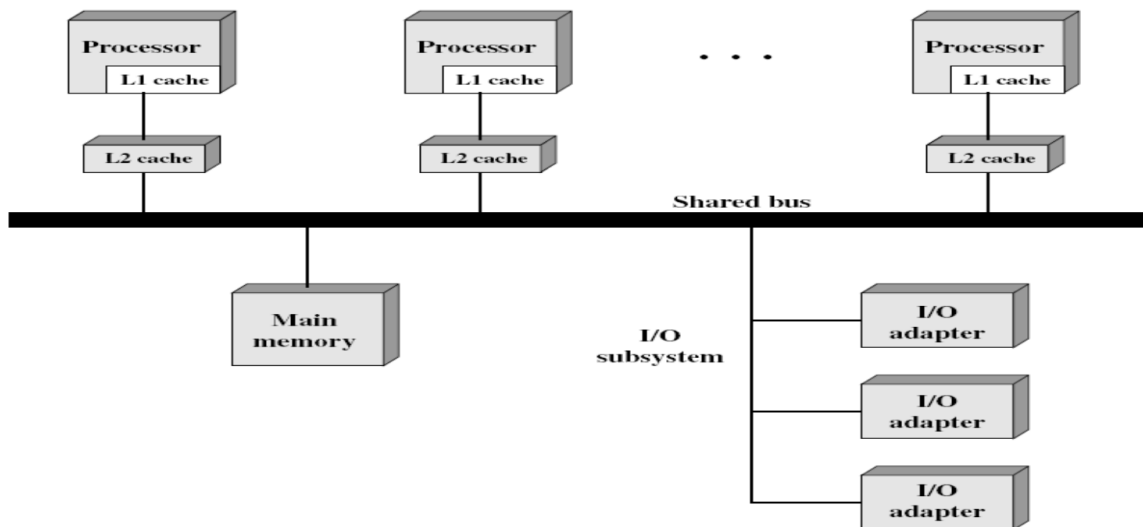


Fig.65. Symmetric multiprocessor organization

The main drawback to the bus organization is performance. All memory references pass through the common bus. Thus, the bus cycle time limits the speed of the system. To improve performance, it is desirable to equip each processor with a cache memory. This should reduce the number of bus accesses dramatically. Typically, workstation and PC SMPs have two levels of cache, with the L1 cache internal (same chip as the processor) and the L2 cache either internal or external. Some processors now employ a L3 cache as well.

The use of caches introduces some new design considerations. Because each local cache contains an image of a portion of memory, if a word is altered in one cache, it could conceivably invalidate a word in another cache. To prevent this, the other processors must be alerted that an update has taken place. This problem is known as the cache coherence problem and is typically addressed in hardware rather than by the operating system.

Cache coherence

In contemporary multiprocessor systems, it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve reasonable performance. It does, however, create a problem known as the *cache coherence* problem. The essence of the problem is this: Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result. There are two common write policies:

- **Write back:** Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.
- **Write through:** All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

It is clear that a write-back policy can result in inconsistency. If two caches contain the same line, and the line is updated in one cache, the other cache will unknowingly have an invalid value. Subsequent reads to that invalid line produce invalid results. Even with the write-through policy,

inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update.

Cache coherence approaches have generally been divided into:

1. Software approaches (Slow).
2. Hardware approaches (Fast).

7.3 Cluster:

An important and relatively recent development computer system design is clustering. Clustering is an alternative to symmetric multiprocessing as an approach to providing high performance and high availability and is particularly attractive for server applications. We can define a cluster as a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster; in the literature, each computer in a cluster is typically referred to as a *node*.

There are four benefits that can be achieved with clustering. These can also be thought of as objectives or design requirements:

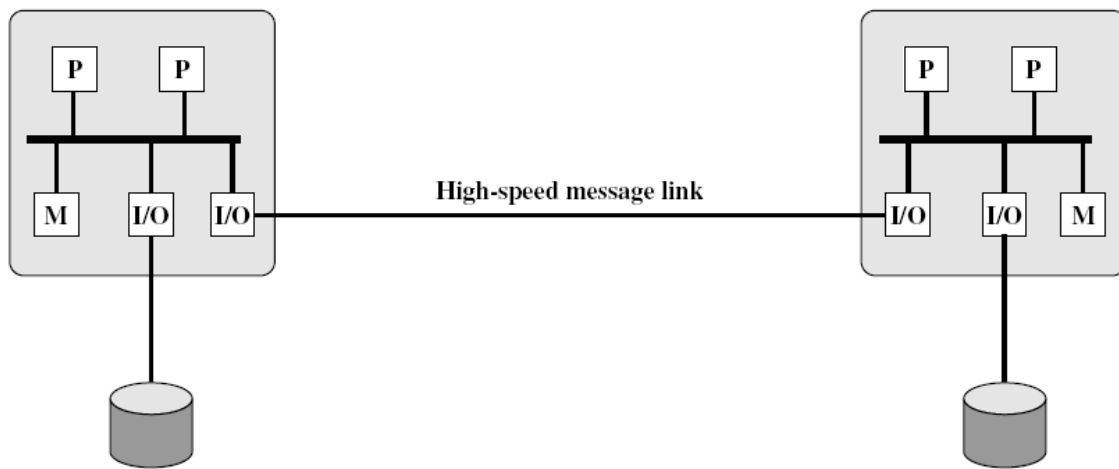
- **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest standalone machines. A cluster can have tens, hundreds, or even thousands of machines, each of which is a multiprocessor.
- **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system.
- **High availability:** Because each node in a cluster is a standalone computer, the failure of one node does not mean loss of service. In many products, fault tolerance is handled automatically in software.
- **Superior price/performance:** By using commodity building blocks,

it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

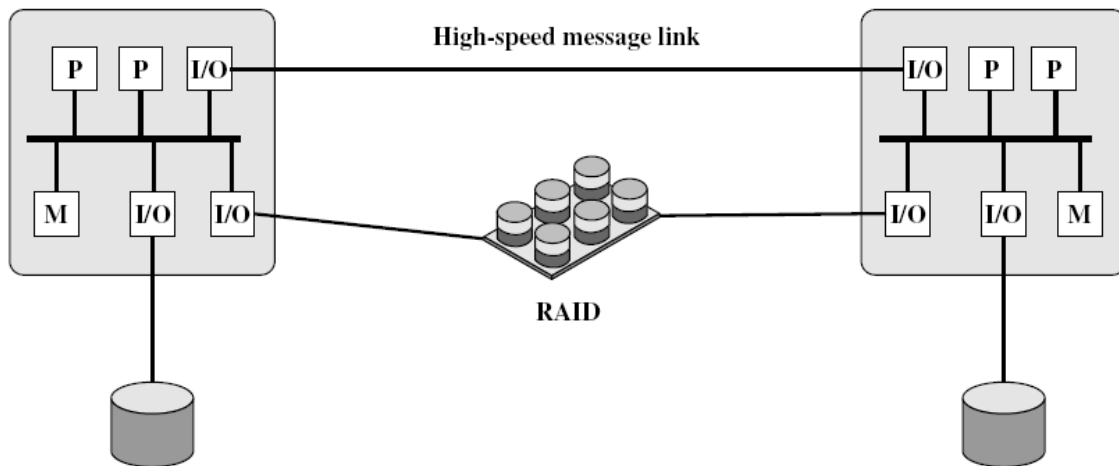
Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Fig.66a shows a two-node cluster in which the only interconnection is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so that there is a connection between the server cluster and remote client systems. Note that in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Fig.66, the other alternative is a shared-disk cluster. In this case, there generally is still a message link between nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In this figure, the common disk subsystem is a RAID system. The use of RAID or some similar redundant disk technology is common in clusters so that the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.



(a) Standby server with no shared disk



(b) Shared Disk

Fig.66. Cluster configurations

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP schemes have been around far longer.

The main strength of the SMP approach is that an SMP is easier to manage and configure than a cluster. The SMP is much closer to the original single-processor model for which nearly all applications are written. The

principal change required in going from a uniprocessor to an SMP is to the scheduler function. Another benefit of the SMP is that it usually takes up less physical space and draws less power than a comparable cluster. A final important benefit is that the SMP products are well established and stable.

Over the long run, however, the advantages of the cluster approach are likely to result in clusters dominating the high-performance server market. Clusters are far superior to SMPs in terms of incremental and absolute scalability. Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

7.4 Nonuniform Memory Access (NUMA)

In terms of commercial products, the two common approaches to providing a multiple-processor system to support applications are SMPs and clusters. For some years, another approach, known as nonuniform memory access (NUMA), has been the subject of research and commercial NUMA products are now available.

Before proceeding, we should define some terms often found in the NUMA literature.

- **Uniform memory access (UMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor to all regions of memory is the same. The access times experienced by different processors are the same. The SMP organization discussed in Section 7.2 is UMA.
- **Nonuniform memory access (NUMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ.
- **Cache-coherent NUMA (CC-NUMA):** A NUMA system in which cache coherence is maintained among the caches of the various processors.

A NUMA system without cache coherence is more or less equivalent to

a cluster. The commercial products that have received much attention recently are CC-NUMA systems, which are quite distinct from both SMPs and clusters. Usually, but unfortunately not always, such systems are in fact referred to in the commercial literature as CC-NUMA systems. This section is concerned only with CC-NUMA systems.

Motivation

With an SMP system, there is a practical limit to the number of processors that can be used. An effective cache scheme reduces the bus traffic between any one processor and main memory. As the number of processors increases, this bus traffic also increases. Also, the bus is used to exchange cache-coherence signals, further adding to the burden. At some point, the bus becomes a performance bottleneck. Performance degradation seems to limit the number of processors in an SMP configuration to somewhere between 16 and 64 processors.

The processor limit in an SMP is one of the driving motivations behind the development of cluster systems. However, with a cluster, each node has its own private main memory; applications do not see a large global memory. In effect, coherency is maintained in software rather than hardware. This memory granularity affects performance and, to achieve maximum performance, software must be tailored to this environment. One approach to achieving large-scale multiprocessing while retaining the flavor of SMP is NUMA.

The objective with NUMA is to maintain a transparent system wide memory while permitting multiple multiprocessor nodes, each with its own bus or other internal interconnect system.

Organization

Fig.67 depicts a typical CC-NUMA organization. There are multiple independent nodes, each of which is, in effect, an SMP organization. Thus, each node contains multiple processors, each with its own L1 and L2 caches, plus main memory. The node is the basic building block of the overall CC-NUMA organization. The nodes are interconnected by means of some communications facility, which could be a switching mechanism, a ring, or some other networking facility.

Each node in the CC-NUMA system includes some main memory. From the point of view of the processors, however, there is only a single addressable memory, with each location having a unique system wide address. When a processor initiates a memory access, if the requested memory location is not in that processor's cache, then the L2 cache initiates a fetch operation. If the desired line is in the local portion of the main memory, the line is fetched across the local bus. If the desired line is in a remote portion of the main memory, then an automatic request is sent out to fetch that line across the interconnection network, deliver it to the local bus, and then deliver it to the requesting cache on that bus. All of this activity is automatic and transparent to the processor and its cache.

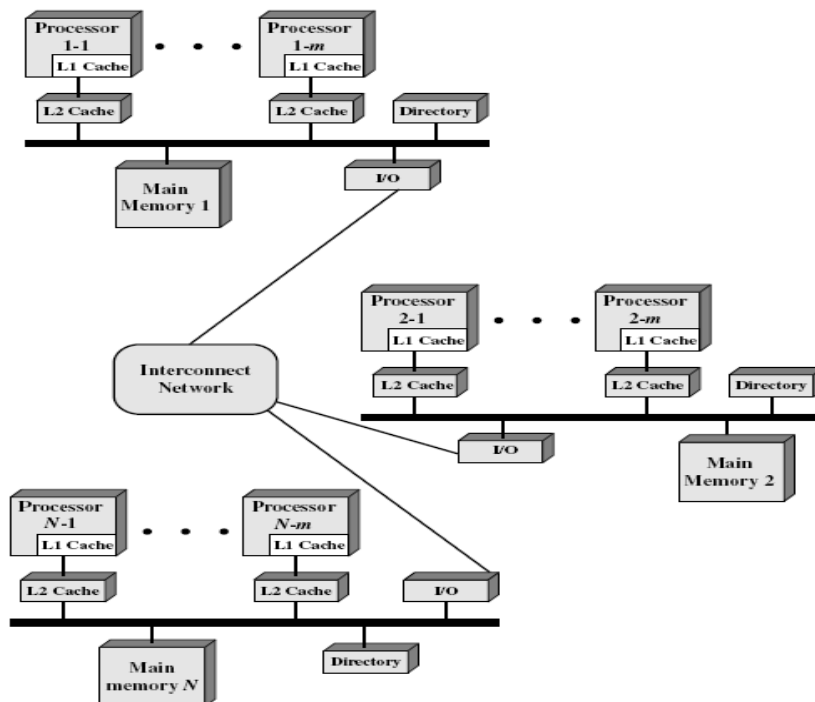


Fig.67. CC-NUMA organization

Problems

- [1] List and briefly define three types of computer system organization.
- [2] What are the chief characteristics of an SMP?
- [3] What are some of the potential advantages of an SMP compared with a uniprocessor?
- [4] What are some of the key benefits of clustering?
- [5] What are the differences among UMA, NUMA, and CC-NUMA?

APPENDIX

Parallelism in microinstructions

Microprogrammable processors are frequently characterized by the maximum number of microoperations that can be specified by a single microinstruction. Microinstructions are often designed to take advantage of the fact that at the microprogramming level, many operations can be performed in parallel. If all useful combinations of parallel microoperations were specified by a single opcode, the number of opcodes would, in most cases, be enormous. Furthermore, a microinstruction decoder of considerable complexity would be needed. To avoid these difficulties it is usual to divide the microoperation specification part of the microinstruction into k disjoint control fields.

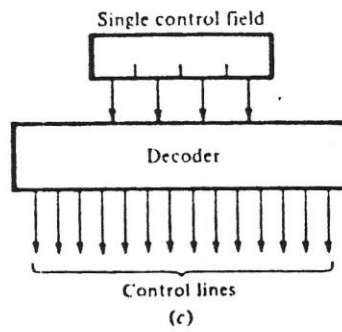
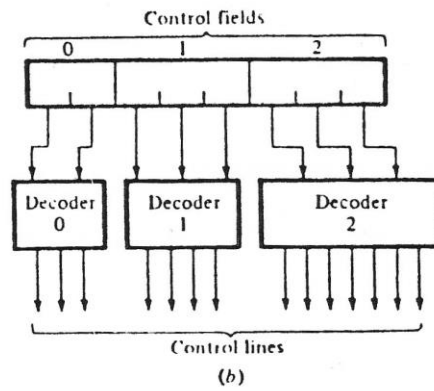
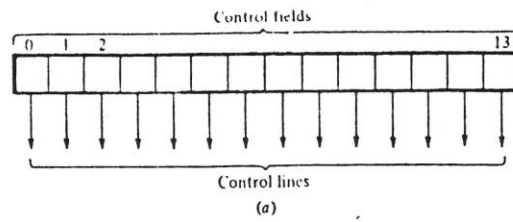
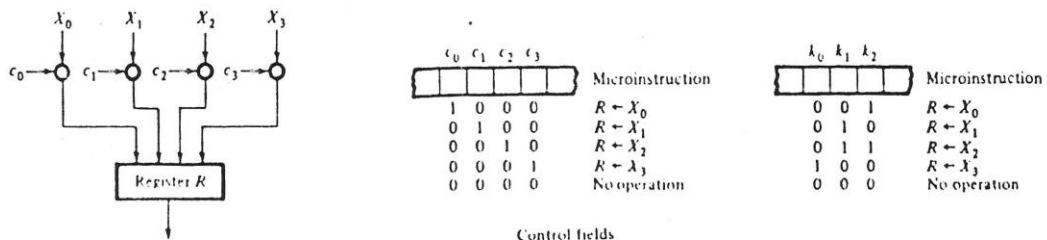
A control field usually specifies the control line values for a single device such as an adder, a register, or a bus. In the extreme case, there may be 1-bit control field for every control line in the system.

Consider, for instance, the register R , which may be loaded from one of four independent sources using the control lines $c_0 - c_3$. Suppose that there is 1-bit for each of these control lines in a microinstruction control field.

In general, any n independent control signals or microoperations can be encoded in a control field of $\lceil \log_2 (n+1) \rceil$ bits, assuming that it is necessary to be able to specify no operation condition when no control signals are to be activated. The unencoded format has the advantage that the control signals may be derived directly from the microinstruction, while using encoded control fields a decoder must be used.

Microinstructions are commonly classified as horizontal or vertical. The horizontal microinstruction have the general attributes: long format, ability to express a high degree of parallelism and little encoding of the control information.

Vertical microinstructions, on the other hand, are characterized by: short formats, limited ability to express parallel microoperations and considerable encoding of the control information. Thus, the format of figure (a) is horizontal, and that of figure (b) is vertical. The case represented by figure (c) may be called horizontal by some authors and vertical by others.



REFERENCES

References:

1. A. Lotze, "Datenverarbeitung I", lectures, Stuttgart University, WS 1975/1976.
2. J. Swoboda, "Datenverarbeitung II", lectures, Stuttgart University, SS 1976.
3. K. Leipold, "Systemprogramme Groessere Rechen-Anlage II", 1976.
4. John P. Hayes, "Digital System Design and Microprocessors", McGraw-Hill, 1984.
5. K. Hwang & F. A. Briggs, "Computer Architecture and Parallel Processing", McGraw-Hill, 1985.
6. Adi J. Khambata, "Microprocessors/Microcomputers Architecture, Software, and Systems", John Wiley & Sons, 1982.
7. M. Morris Mano, "Computer Engineering: Hardware Design", Prentice Hall, 1988.
8. Taylor L. Booth, "Introduction to Computer Engineering Hardware and Software Design", John Wiley & Sons, 1984.
9. John P. Hayes, "Computer Architecture and Organization", McGraw-Hill, 1988.
10. V. Carl Hamacher, Z. G. Vranesic & Safwat G. Zaky, "Computer Organization", McGraw-Hill, 1984.
11. L. H. Pollard, "Computer Design and Architecture", Prentice-Hall, 1990.

12. Amin K. Ismail & Victor M. Kooney, “Microprocessor Hardware and Software Concepts”, Macmillan Publishing Company & Collier Macmillan Publisher, 1987.
13. D. A. Protopapas, “Microcomputer Hardware Design”, Prentice-Hall International Edition, 1988.
14. Thomas R. Mccalla, “Digital Logic and Computer Design”, Maxwell Macmillan Intern. Edition, 1992.
15. William Stallings, “Computer Organization and Architecture”, Pearson Prentice Hall, Eighth edition, 2010.